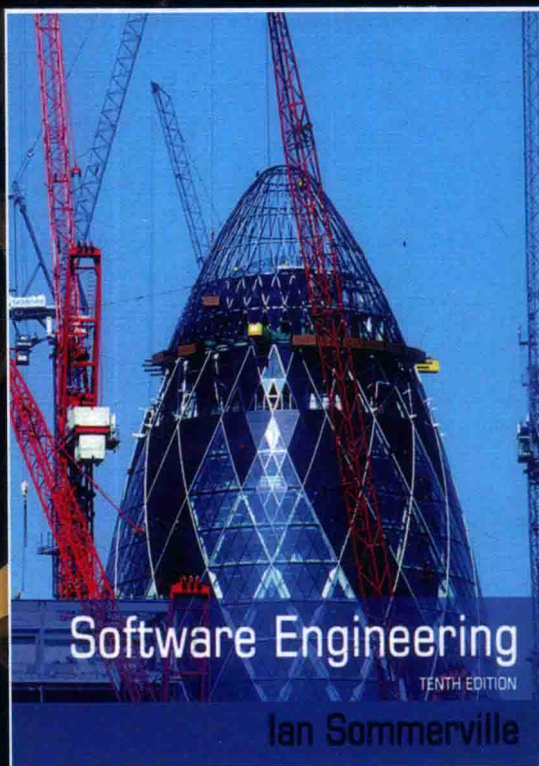


软件工程

[英] 伊恩·萨默维尔 (Ian Sommerville) 著

彭鑫 赵文耘 等译

Software Engineering
Tenth Edition



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

原书第10版

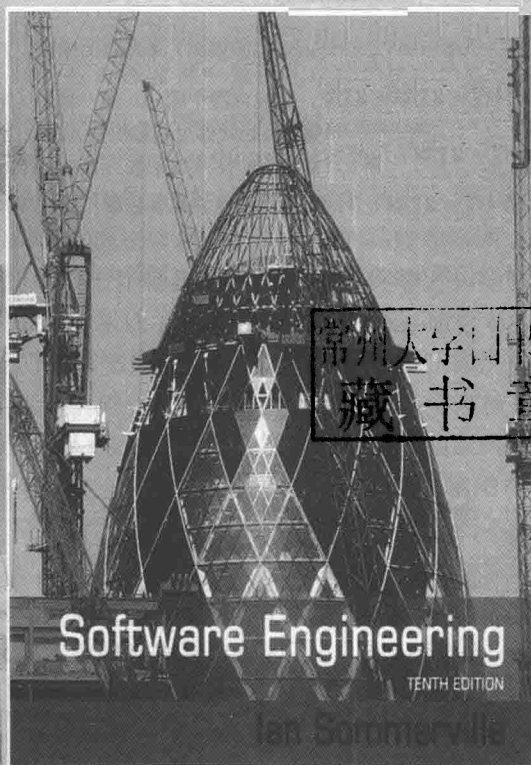
软件工程

[英] 伊恩·萨默维尔 (Ian Sommerville) 著

彭鑫 赵文耘 等译

Software Engineering

Tenth Edition



机械工业出版社
China Machine Press

图书在版编目(CIP)数据

软件工程(原书第10版)/(英)伊恩·萨默维尔(Ian Sommerville)著;彭鑫等译. —北京:机械工业出版社,2018.1

(计算机科学丛书)

书名原文:Software Engineering, Tenth Edition

ISBN 978-7-111-58910-5

I. 软… II. ①伊… ②彭… III. 软件工程—高等学校—教材 IV. TP311.5

中国版本图书馆CIP数据核字(2018)第003918号

本书版权登记号:图字 01-2015-2931

Authorized translation from the English language edition, entitled *Software Engineering, Tenth Edition*, 978-0-13-394303-0 by Ian Sommerville, published by Pearson Education, Inc., Copyright © 2016, 2011, 2006.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press Copyright © 2018.

本书中文简体字版由Pearson Education(培生教育出版集团)授权机械工业出版社在中华人民共和国境内(不包括香港、澳门特别行政区及台湾地区)独家出版发行。未经出版者书面许可,不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。

本书是系统介绍软件工程理论的经典教材,共包含四个部分:第一部分(第1~9章)是对软件工程的一般性介绍,介绍了软件工程的一些重要概念(如软件过程和敏捷方法),描述了基本的软件开发活动(从需求规格说明一直到系统演化);第二部分(第10~14章)关注软件系统可依赖性和信息安全等重要话题;第三部分(第15~21章)介绍更高级的软件工程话题;第四部分(第22~25章)关注技术管理问题。

本书适合作为软件和系统工程专业本科生或研究生教材,同时也是软件工程师难得的参考书。

出版发行:机械工业出版社(北京市西城区百万庄大街22号 邮政编码:100037)

责任编辑:迟振春

责任校对:李秋荣

印刷:中国电影出版社印刷厂

版次:2018年2月第1版第1次印刷

开本:185mm×260mm 1/16

印张:33.25

书号:ISBN 978-7-111-58910-5

定价:89.00元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线:(010) 88378991 88361066

投稿热线:(010) 88379604

购书热线:(010) 68326294 88379649 68995259

读者信箱:hjzsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问:北京大成律师事务所 韩光/邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅肇划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum, Bjarne Stroustrup, Brain W. Kernighan, Dennis Ritchie, Jim Gray, Afred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

很荣幸能从北京理工大学程成老师手中接过本书的翻译任务。程成老师从第6版开始承担本书的翻译任务，直至第9版，为我们这次第10版的翻译打下了良好的基础。

作为一本经典的软件工程教材，本书的内容非常全面。其中，第一部分覆盖了基本的软件开发生命周期，可以作为本科软件工程课程教学的基本内容。第二部分针对高可信软件系统对于可依赖性和信息安全的高要求，介绍了相应的软件工程方法和技术，体现了软件工程对于大规模复杂软件系统的质量保障作用。第三和第四部分则分别针对软件工程中的一些高级技术问题和开发管理问题进行了介绍。后三个部分的内容为面向高年级本科生和研究生的高级软件工程课程以及软件开发管理、软件可靠性工程等其他更深入的软件工程专业课程提供了教学内容。

难能可贵的是，本书每一次改版都能根据软件工程方法、技术及产业实践的最新发展更新并调整相关内容。本版在第9版的基础上，更新了关于敏捷软件工程的相关内容，增加了RESTful 服务、系统之系统等反映软件开发技术和复杂系统发展趋势的新内容，重新组织了与高可信软件系统密切相关的多个章节，同时将面向方面的软件工程以及过程改进等不太重要的章节移到了网站上。

本书系统反映了工程化软件开发所具有的严谨性和规范性，但同时并不教条。例如，虽然系统性地介绍了UML 建模方法，但在谈到UML 在软件设计过程中的作用时，作者指出非正式的表达法在软件设计过程中可能更有用。因此，我们在学习本书所介绍的软件工程方法和规范的同时，也不要忘了与软件开发实践相结合，在做中学，在实践中领会。

本书主要由彭鑫、赵文耘翻译。参与本书翻译工作的还有复旦大学计算机科学技术学院研究生刘汶淦、周翔、姜清涛、陈驰、黄凯峰等。此外，机械工业出版社华章公司的朱劼、迟振春两位编辑还认真审校了译稿，在此一并表示感谢。

由于时间仓促以及译者自身知识和水平有限，译稿中难免存在错误和遗漏，在此向广大读者表示歉意，敬请批评指正！

软件工程在过去 50 年之中取得了令人瞩目的进展。我们的社会已经无法在缺少大型专业化软件系统的情况下正常运转了。国家的公用事业和基础设施（如能源、通信、交通）全都依赖于复杂且可靠的计算机系统。软件使得我们可以探索空间，创造万维网这一人类历史上最重要的信息系统。智能手机和平板电脑无处不在，而为这些设备开发软件的整个“应用开发产业”已经在过去几年中悄然形成。

人类现在正面临着一系列迫切的挑战——气候变化和极端天气、自然资源的减少、需要为更多的人口提供食物和住房、国际恐怖主义的威胁，以及为老年人提供令人满意的生活。我们需要新技术来帮助我们应对这些挑战，可以肯定的是软件将在这些技术中扮演核心角色。因此，软件工程对于我们在这个星球上的未来极其重要。我们必须继续培养软件工程师并推动软件工程学科的持续发展，从而满足开发更多的软件系统以及创造我们所需要的越来越复杂的未来系统的需要。

当然，软件项目还存在很多问题。系统仍然有时会延迟交付并且成本超支。我们正在创造越来越复杂的软件系统之系统（software systems of systems），在这条道路上遇到各种困难也是不足为奇的。然而，我们不应该让这些问题掩盖软件工程领域已经取得的巨大成就，以及所形成的各种令人印象深刻的软件工程方法和技术。

本书的不同版本已经有超过 30 年的历史，而这一版同样遵循了本书第 1 版中所建立的基本原则：

1. 按照工业界实践介绍软件工程，不对任何特定的方法（例如敏捷开发、形式化方法）持倾向性态度。在现实中，工业界往往将各种技术（例如敏捷以及基于计划的开发）混合在一起使用，这一点也在本书中有所反映。

2. 根据我所知道的以及所理解的知识介绍软件工程。很多人建议我增加并详细介绍其他相关话题，例如开源软件开发、UML 的使用以及移动软件工程等，但是我对这些领域的了解并不多，我个人的工作主要是在系统可靠性和系统工程方面，这一点在我为本书所选择的高级专题中有所反映。

我认为现代软件工程的关键问题是管理复杂性，将敏捷和其他方法结合起来，并确保我们的系统安全以及有韧性。这些问题是我在这一版中修改和新增内容的主要因素。

对第 9 版的修改

这一版相比第 9 版的更新和新增内容汇总如下：

- 全面更新了关于敏捷软件工程的章节，增加了关于 Scrum 的新内容。此外还根据需要对其他章节进行了更新，以反映敏捷方法在软件工程中日益增长的应用。
- 增加了关于韧性工程、系统工程、系统之系统的新章节。
- 对于涉及可靠性、安全、信息安全的 3 章进行了彻底的重新组织。
- 在第 18 章“面向服务的软件工程”中增加了关于 RESTful 服务的新内容。
- 更新和修改了关于配置管理的章节，增加了关于分布式版本控制系统的新内容。
- 将关于面向方面的软件工程以及过程改进的章节移到了本书的配套网站上。
- 在网站上新增了补充材料，包括一系列支持视频。我在视频中对于一些关键话题进

行了解释,并且推荐了相关的 YouTube 视频。

这一版保留了此前版本中的四部分结构,但我对其中每个部分都进行了大量的修改。

1. 在第一部分软件工程导论中,我彻底重写了第 3 章(敏捷方法)并对其进行了更新,以反映 Scrum 方法在实践中日益增长的使用。第 1 章增加了一个关于数字化学习环境的案例研究,这个案例在其他几个章节中也会用到。第 9 章更加详细地介绍了遗留系统。这一部分的其他章节也都进行了少量的修改和更新。

2. 第二部分介绍系统可依赖性。这一部分进行了修改和重新组织,不再按照面向活动的方式进行组织,而是将安全、信息安全、可靠性分散在多个章节中。这使得相关内容(例如信息安全)可以更加方便地作为独立的专题在更加综合性的课程中使用。我增加了关于韧性工程的一章,其中涉及网络安全、组织韧性以及韧性系统设计。

3. 第三部分增加了关于系统工程、系统之系统的新章节,并对与面向服务的系统工程相关的内容进行了全面修改,以反映 RESTful 服务的使用日益增长的趋势。与面向方面的软件工程相关的章节移到了本书的配套网站上。

4. 第四部分对配置管理的内容进行了更新,以反映分布式版本控制工具(如 Git)的使用日益增长的趋势。过程改进相关的章节移到了本书的配套网站上。

本书补充材料中的一个重要变化是为每个章节增加了视频推荐。我制作了关于一系列主题的 40 多段视频,放在我的 YouTube 频道上并可从本书的网页上链接过去。对于没有制作视频的地方,我推荐了一些可能有用的 YouTube 视频。

我在下面这个视频中解释了这一版中所做修改背后的原因。

<http://software-engineering-book/videos/10th-edition-changes>

读者对象

本书主要面向各大学和学院正在学习软件和系统工程初高级课程的学生。我假设读者对于编程基础和基本数据结构都已有所理解。

工业界的软件工程师也会发现这是一本很好的读物,能帮助他们在软件复用、体系结构设计、可依赖性和信息安全性以及系统工程等方面获得新的知识。

教学建议

针对三种不同类型的软件工程课程,我对本书进行了如下设计:

- 软件工程一般导论课程。本书的第一部分专门用于一个学期的软件工程专业导论课程。这部分包括 9 章,涵盖了软件工程领域的基础内容。如果你的课程中包含实践性教学环节,那么可以选讲第四部分中关于管理的章节。
- 软件工程特定主题的导论课程或进阶课程。通过使用本书第二~四部分的内容,可以创建一系列更高级的课程。例如,我采用第二部分的各章加上关于系统工程和质量管理的两章来讲授以关键系统为主题的课程。而对于讨论软件密集型系统的课程,我选择的章节涉及系统工程、需求工程、系统之系统、分布式软件工程、嵌入式软件、项目管理和项目计划。
- 软件工程特定主题的更高阶课程。对于这类课程,本书的各章可以构成课程的基础,然后辅之以更多的阅读以便进一步探索某个主题。例如,关于软件复用的课程就可以基于第 15~18 章的内容。

用书教师可以访问培生网站获取相关教辅资源,网址为 <http://www.pearsonhighered.com/>

sommerville^①。部分资源为加密内容,可在网站上通过注册来获取密码。教辅资源包括:

- 部分章末练习题的答案。
- 每章的测验题目及答案。

配套网站

本书采用印刷和在线网站内容相结合的方式,其中印刷版中的核心信息可以链接到网站上的补充材料。有些章节包含特别编写的在线段落以提供更多的信息。在线网站上还有 6 个在线章节,介绍了若干我在本书的印刷版中未介绍的主题。

读者可以从本书的网站(software-engineering-book.com)上下载丰富的补充材料,包括:

- 本书所有章节的 PPT;
- 我所录制的针对一系列软件工程主题的视频,我还推荐了一些有助于学习本书内容的 YouTube 视频;
- 针对课程教师的指南,其中给出了在教授不同课程时如何使用本书的建议;
- 关于本书中案例研究(胰岛素泵、心理健康保健系统、野外气象站系统、数字化学习系统)的附加信息,以及其他一些案例研究(例如阿丽亚娜 5 型运载火箭失效);
- 6 个在线章节,介绍了过程改进、形式化方法、交互设计、应用体系结构、文档化以及面向方面的开发;
- 为每个章节提供补充内容的在线段落,这些在线段落可以通过每一章中用方框突出显示的链接来访问;
- 附加的涉及一系列系统工程主题的 PPT。

应本书读者的要求,我已经在本书的网站上发布了其中一个系统案例研究的完整需求规格说明。对于学生而言,获得这种文档并理解其结构和复杂性有些困难。为了避免保密问题,我对这个来自于真实系统的需求文档进行了再工程,因此使用该文档没有任何限制。

联系信息

网站: software-engineering-book.com

邮件: 名字, software.engineering.book@gmail.com

博客: iansommerville.com/systems-software-and-technology

YouTube: youtube.com/user/SoftwareEngBook

Facebook: facebook.com/sommerville.software.engineering

Twitter: @SoftwareEngBook 或者 @iansommerville (获取更广泛的推文)

请在 Twitter 或 Facebook 上关注我,以便及时获得关于软件和系统工程的新材料和新评论。

致谢

多年以来许多人都对本书的演进做出了贡献,在此我想感谢每一位曾经评论过本书此前版本并且给出了建设性意见的人(审阅人、学生、读者)。我想特别感谢我的家人 Anne、Ali 和 Jane,感谢他们在我编写本书(以及此前所有版本)过程中对我的爱、帮助和支持。

Ian Sommerville

2014 年 9 月

① 关于本书教辅资源,只有使用本书作为教材的教师才可以申请,需要的教师请联系机械工业出版社华章公司,电话 136 0115 6823,邮箱 wanguang@hzbook.com。——编辑注

目 录

Software Engineering, Tenth Edition

出版者的话

译者序

前言

第一部分 软件工程导论

第1章 概述 2

1.1 专业化软件开发 3

1.1.1 软件工程 5

1.1.2 软件工程的多样性 6

1.1.3 互联网软件工程 8

1.2 软件工程职业道德 9

1.3 案例研究 11

1.3.1 胰岛素泵控制系统 12

1.3.2 心理健康治疗病人信息系统 13

1.3.3 野外气象站 14

1.3.4 学校数字化学习环境 15

要点 16

阅读推荐 17

网站 17

练习 17

参考文献 18

第2章 软件过程 19

2.1 软件过程模型 20

2.1.1 瀑布模型 21

2.1.2 增量式开发 23

2.1.3 集成与配置 25

2.2 过程活动 26

2.2.1 软件规格说明 26

2.2.2 软件设计和实现 27

2.2.3 软件确认 29

2.2.4 软件演化 30

2.3 应对变化 31

2.3.1 原型 32

2.3.2 增量式交付 33

2.4 过程改进 34

要点 36

阅读推荐 36

网站 37

练习 37

参考文献 37

第3章 敏捷软件开发 39

3.1 敏捷方法 40

3.2 敏捷开发技术 42

3.2.1 用户故事 43

3.2.2 重构 44

3.2.3 测试先行的开发 45

3.2.4 结对编程 46

3.3 敏捷项目管理 47

3.4 敏捷方法的伸缩 50

3.4.1 敏捷方法的实践问题 51

3.4.2 敏捷和计划驱动的方法 52

3.4.3 面向大型系统的敏捷方法 54

3.4.4 面向整个组织的敏捷方法 56

要点 57

阅读推荐 57

网站 58

练习 58

参考文献 58

第4章 需求工程 60

4.1 功能性需求和非功能性需求 62

4.1.1 功能性需求 63

4.1.2 非功能性需求 64

4.2 需求工程过程 66

4.3 需求抽取 67

4.3.1 需求抽取技术 69

4.3.2 故事和场景	71	6.3.1 分层体系结构	111
4.4 需求规格说明	73	6.3.2 知识库体系结构	112
4.4.1 自然语言规格说明	74	6.3.3 客户-服务器体系结构	113
4.4.2 结构化规格说明	75	6.3.4 管道和过滤器体系结构	115
4.4.3 用况	76	6.4 应用体系结构	116
4.4.4 软件需求文档	77	6.4.1 事务处理系统	117
4.5 需求确认	79	6.4.2 信息系统	118
4.6 需求变更	80	6.4.3 语言处理系统	119
4.6.1 需求管理计划	81	要点	121
4.6.2 需求变更管理	82	阅读推荐	122
要点	83	网站	122
阅读推荐	83	练习	122
网站	84	参考文献	123
练习	84		
参考文献	85		
第5章 系统建模	86	第7章 设计和实现	125
5.1 上下文模型	87	7.1 使用UML的面向对象设计	126
5.2 交互模型	89	7.1.1 系统上下文和交互	126
5.2.1 用况建模	90	7.1.2 体系结构设计	128
5.2.2 顺序图	91	7.1.3 对象类识别	129
5.3 结构模型	93	7.1.4 设计模型	130
5.3.1 类图	93	7.1.5 接口规格说明	133
5.3.2 泛化	95	7.2 设计模式	134
5.3.3 聚集	95	7.3 实现问题	136
5.4 行为模型	96	7.3.1 复用	137
5.4.1 数据驱动的建模	96	7.3.2 配置管理	138
5.4.2 事件驱动的建模	97	7.3.3 宿主机-目标机开发	139
5.4.3 模型驱动的工程	99	7.4 开源开发	141
5.5 模型驱动的体系结构	100	7.4.1 开源许可证	142
要点	102	要点	143
阅读推荐	103	阅读推荐	143
网站	103	网站	144
练习	103	练习	144
参考文献	104	参考文献	145
第6章 体系结构设计	105	第8章 软件测试	146
6.1 体系结构设计决策	107	8.1 开发测试	149
6.2 体系结构视图	109	8.1.1 单元测试	150
6.3 体系结构模式	110	8.1.2 选择单元测试用例	151
		8.1.3 构件测试	153
		8.1.4 系统测试	155

8.2 测试驱动的开发	157	第 11 章 可靠性工程	200
8.3 发布测试	159	11.1 可用性和可靠性	202
8.3.1 基于需求的测试	159	11.2 可靠性需求	203
8.3.2 场景测试	160	11.2.1 可靠性度量	204
8.3.3 性能测试	161	11.2.2 非功能性可靠性需求	205
8.4 用户测试	161	11.2.3 功能性可靠性规格说明	207
要点	163	11.3 容错体系结构	207
阅读推荐	164	11.3.1 保护性系统	208
网站	164	11.3.2 自监控系统体系结构	208
练习	164	11.3.3 <i>N</i> 版本编程	210
参考文献	165	11.3.4 软件多样性	211
第 9 章 软件演化	166	11.4 可靠性编程	212
9.1 演化过程	168	11.5 可靠性度量	216
9.2 遗留系统	170	11.5.1 运行概况	218
9.2.1 遗留系统管理	173	要点	218
9.3 软件维护	176	阅读推荐	219
9.3.1 维护预测	178	网站	219
9.3.2 软件再工程	179	练习	219
9.3.3 软件重构	181	参考文献	220
要点	182	第 12 章 安全工程	221
阅读推荐	182	12.1 安全关键系统	222
网站	183	12.2 安全需求	224
练习	183	12.2.1 危险识别	225
参考文献	183	12.2.2 危险评估	226
第二部分 系统可依赖性和信息安全		12.2.3 危险分析	227
第 10 章 可依赖系统	186	12.2.4 风险降低	229
10.1 可依赖性属性	187	12.3 安全工程过程	229
10.2 社会技术系统	189	12.3.1 安全保证过程	230
10.2.1 规章与守约	191	12.3.2 形式化验证	232
10.3 冗余和多样性	192	12.3.3 模型检测	233
10.4 可依赖的过程	193	12.3.4 静态程序分析	234
10.5 形式化方法与可依赖性	195	12.4 安全案例	235
要点	197	12.4.1 结构化论证	237
阅读推荐	198	12.4.2 软件的安全论证	238
网站	198	要点	240
练习	198	阅读推荐	241
参考文献	199	网站	241
		练习	241
		参考文献	242

第 13 章 信息安全工程	244
13.1 信息安全和可依赖性	245
13.2 信息安全和组织	248
13.2.1 信息安全风险评估	249
13.3 信息安全需求	249
13.3.1 滥用案例	252
13.4 信息安全系统设计	253
13.4.1 设计风险评估	254
13.4.2 体系结构设计	257
13.4.3 设计准则	259
13.4.4 信息安全系统编程	263
13.5 信息安全测试和保证	263
要点	265
阅读推荐	265
网站	265
练习	266
参考文献	266

第 14 章 韧性工程	268
14.1 网络安全	270
14.2 社会技术韧性	273
14.2.1 人为错误	275
14.2.2 运行和管理过程	277
14.3 韧性系统设计	279
要点	284
阅读推荐	284
网站	285
练习	285
参考文献	285

第三部分 高级软件工程

第 15 章 软件复用	288
15.1 复用概览	290
15.2 应用框架	292
15.3 软件产品线	294
15.4 应用系统复用	298
15.4.1 可配置的应用系统	299
15.4.2 集成的应用系统	301
要点	303

阅读推荐	303
网站	304
练习	304
参考文献	305

第 16 章 基于构件的软件工程	306
16.1 构件和构件模型	307
16.1.1 构件模型	310
16.2 CBSE 过程	311
16.2.1 面向复用的 CBSE	312
16.2.2 基于复用的 CBSE	314
16.3 构件组装	316
要点	321
阅读推荐	321
网站	322
练习	322
参考文献	322

第 17 章 分布式软件工程	324
17.1 分布式系统	325
17.1.1 交互模型	327
17.1.2 中间件	329
17.2 客户-服务器计算	330
17.3 分布式系统的体系结构模式	331
17.3.1 主从体系结构	332
17.3.2 两层客户-服务器体系结构	332
17.3.3 多层客户-服务器体系结构	334
17.3.4 分布式构件体系结构	335
17.3.5 对等体系结构	338
17.4 软件即服务	340
要点	342
阅读推荐	343
网站	343
练习	343
参考文献	344

第 18 章 面向服务的软件工程	345
18.1 面向服务的体系结构	348

18.1.1 SOA 中的服务构件	349	要点	404
18.2 RESTful 服务	351	阅读推荐	405
18.3 服务工程	354	网站	405
18.3.1 可选服务识别	354	练习	405
18.3.2 服务接口设计	356	参考文献	406
18.3.3 服务实现和部署	358		
18.4 服务组合	359	第 21 章 实时软件工程	407
18.4.1 workflow 设计与实现	361	21.1 嵌入式系统设计	408
18.4.2 服务组合测试	363	21.1.1 实时系统建模	411
要点	364	21.1.2 实时编程	413
阅读推荐	364	21.2 实时软件体系结构模式	413
网站	365	21.2.1 观察和反应模式	414
练习	365	21.2.2 环境控制模式	415
参考文献	366	21.2.3 处理管道模式	417
第 19 章 系统工程	367	21.3 时序分析	418
19.1 社会技术系统	370	21.4 实时操作系统	421
19.1.1 涌现特性	372	21.4.1 进程管理	422
19.1.2 不确定性	373	要点	423
19.1.3 成功准则	374	阅读推荐	424
19.2 概念设计	375	网站	424
19.3 系统采购	377	练习	424
19.4 系统开发	380	参考文献	425
19.5 系统运行和演化	383		
19.5.1 系统演化	384	第四部分 软件管理	
要点	385	第 22 章 项目管理	428
阅读推荐	385	22.1 风险管理	430
网站	385	22.1.1 风险识别	431
练习	386	22.1.2 风险分析	432
参考文献	386	22.1.3 风险计划	433
第 20 章 系统之系统	388	22.1.4 风险监控	434
20.1 系统复杂度	390	22.2 人员管理	435
20.2 系统之系统的分类	393	22.2.1 激励人员	435
20.3 还原论和复杂系统	395	22.3 团队协作	437
20.4 系统之系统工程	397	22.3.1 成员的挑选	439
20.4.1 接口开发	398	22.3.2 小组的结构	440
20.4.2 集成和部署	399	22.3.3 小组的沟通	442
20.5 系统之系统的体系结构	400	要点	443
20.5.1 系统之系统的体系结构模式	402	阅读推荐	443
		网站	443

练习	444	24.2.1 ISO 9001 标准框架	473
参考文献	444	24.3 评审与审查	475
第 23 章 项目计划	445	24.3.1 评审过程	475
23.1 软件报价	446	24.3.2 程序审查	476
23.2 计划驱动的开发	447	24.4 质量管理与敏捷开发	478
23.2.1 项目计划	448	24.5 软件度量	479
23.2.2 计划过程	449	24.5.1 产品量度	481
23.3 项目进度安排	450	24.5.2 软件构件分析	483
23.3.1 进度安排表示方法	451	24.5.3 度量歧义	484
23.4 敏捷计划	453	24.5.4 软件解析	485
23.5 估算技术	455	要点	486
23.5.1 算法成本建模	456	阅读推荐	486
23.6 COCOMO 成本建模	458	网站	487
23.6.1 应用组合模型	459	练习	487
23.6.2 早期设计模型	460	参考文献	487
23.6.3 复用模型	460	第 25 章 配置管理	489
23.6.4 后体系结构模型	462	25.1 版本管理	492
23.6.5 项目的工期和人员配备	464	25.2 系统构建	495
要点	464	25.3 变更管理	499
阅读推荐	465	25.4 发布版本管理	503
网站	465	要点	505
练习	465	阅读推荐	505
参考文献	467	网站	506
第 24 章 质量管理	468	练习	506
24.1 软件质量	469	参考文献	506
24.2 软件标准	471	术语表	507

软件工程导论

这一部分的目标是对软件工程做一般性概述，各章可以用于一个学期的软件工程导论性质的课程。这部分介绍软件工程的一些重要概念（如软件过程和敏捷方法），描述基本的软件开发活动（从需求规格说明一直到系统演化）。

第1章是一个概述，介绍了专业软件工程并定义了一些软件工程的观念。这一章还就软件工程职业道德方面的问题进行了简要的讨论。软件工程师认真思考所从事的工作的深远影响是很重要的。这一章还介绍了本书中所使用的4个案例研究，分别是：管理接受心理健康问题治疗的病人记录的信息系统（Mentcare），便携式胰岛素泵控制系统，野外气象站嵌入式系统，数字化学习环境（iLearn）。

第2章和第3章分别介绍软件工程过程和敏捷开发。第2章介绍软件过程模型，例如瀑布模型，以及这些过程中的基本活动。第3章补充了一个关于软件工程的敏捷开发方法的讨论，并在上一版的基础上进行了全面的修改，主要关注使用Scrum的敏捷开发以及对于敏捷实践的讨论（例如，用于需求定义的故事以及测试驱动开发）。

这部分的其他各章是对第2章中所介绍的软件过程活动的深入阐述。第4章介绍需求工程这一十分重要的话题，其中定义了系统应该做什么的需求。第5章介绍如何使用UML对系统进行建模，主要关注使用用例图、类图、顺序图 and 状态图来建模软件系统。第6章对软件体系结构的重要性进行了讨论，并介绍了如何在软件设计中使用体系结构模式。

第7章介绍面向对象的设计和模式的使用，还介绍了重要的实现问题——复用、配置管理、宿主-目标机开发，并且讨论了开源软件开发。第8章主要介绍软件测试，从系统开发时的单元测试到软件的发布测试。此外，第8章还讨论了测试驱动开发的使用，这是在敏捷方法中率先使用的一种方法，现在已经得到广泛应用。最后，第9章简要介绍了软件演化问题，包括演化过程、软件维护和遗留系统管理。

概述

目标

本章的目标是介绍软件工程的概
念，并为理解本书其他部分内容提供一个框架。阅读完本章后，你将：

- 理解软件工程是什么，为什么它很重要；
- 理解开发不同类型的软件系统可能需要不同的软件工程技术；
- 理解对于软件工程师很重要的道德和职业问题；
- 了解4个不同类型的软件系统，全书都将以它们为例。

软件工程对于政府、社会、国内和国际企业、机构的正常运转都是至关重要的。现在的世界离开软件就无法运转了。国家基础设施和公共服务都是由基于计算机的系统控制的，大多数电子产品都包括计算机和控制软件。工业制造和分销已经完全计算机化了，金融系统也是这样。娱乐业，包括音乐产业、计算机游戏产业、电影和电视产业，也是一个软件密集型的产业。全世界75%以上的人口都有一台软件控制的移动电话，到2016年其中的大多数都能够连接互联网。

软件是抽象的、不可触摸的，它不受物质材料的限制，也不受物理定律或加工过程的制约：一方面使软件工程得以简化，因为软件的潜能不受物理因素的限制；另一方面，由于缺乏自然约束，软件系统也就很容易变得极为复杂，理解它会很困难，改变它价格高昂。

有很多不同类型的软件系统，从简单的嵌入式系统到复杂的全球范围的信息系统。没有放之四海而皆准的软件工程表示法、方法或技术，因为不同类型的软件需要不同的方法。开发一个组织的信息系统和开发一个科学仪器的控制器是完全不同的，而这些系统都跟图形密集型的计算机游戏没有太多的共同点。所有这些应用都需要软件工程，但并不都是需要相同的软件工程方法和技术。

现在仍有许多有关软件项目出问题和“软件失效”的报道。软件工程因不能充分支持现代软件的开发而遭受批评。然而，在我看来，这些所谓的软件失效很多都源于以下两方面的因素：

1. 不断增长的系统复杂性。随着新的软件工程技术可以帮助我们构建更大、更复杂的系统，要求在发生变化。系统必须更快地构建和交付；需要更大甚至更复杂的系统；系统必须具备在以前看来不可能实现的功能。必须不断发展新的软件工程技术来迎接交付更复杂的软件的新挑战。

2. 未有效采用软件工程方法。不采用软件工程的方法和技术去编写计算机程序是相当容易的。许多公司都是随着他们的产品和服务的逐步发展而逐渐陷入软件开发中的。他们在日常工作中不使用软件工程方法，结果导致他们的软件经常比预计的费用高而可靠性低于预期。我们需要更好的软件工程教育和训练来应对这一问题。

软件工程师应该为自己所取得的成就而感到自豪。虽然我们在开发复杂软件时还存在一些问题,但如果没有软件工程,我们就不能探索太空,无法使用互联网和现代的电信服务,而各种形式的旅行则会更加危险和昂贵。人类在 21 世纪所面临的挑战是气候变化、自然资源不足、人口结构变化以及世界人口膨胀。我们将依赖软件工程来开发应对这些问题所需的系统。



软件工程的历史

“软件工程”这一概念是在 1968 年召开的一个讨论所谓的“软件危机”问题的会议上首次被提出的 (Naur and Randell 1969)。当时,人们认识到单一的程序开发技术已经不能适应大型复杂软件系统的开发。这些软件项目经常不可靠、费用高于预期,并且延迟交付。

20 世纪七八十年代,各种新的软件工程技术和方法陆续出现,例如结构化编程、信息隐藏和面向对象开发。各种工具和标准的表示法陆续出现,并逐渐成为现代软件工程的基础。

<http://software-engineering-book.com/web/history/>

1.1 专业化软件开发

许多人都在编写程序。业务人员编写电子表格程序来简化工作;科学家和工程师编写程序来处理实验数据;业余爱好者出于自己的兴趣和爱好而编写程序。然而,大多数软件开发都是一种专业化的活动,软件的开发是为了满足某种业务目标,为了嵌入其他设备中,或者作为软件产品(例如信息系统、计算机辅助设计系统)。专业化开发与个人化开发的关键区别在于,专业化软件除了开发者之外还有其他用户会使用,而且专业化软件通常都是由团队而非个人开发的。专业化软件在其生命周期内要不断维护和修改。

软件工程的目的是支持专业化的软件开发,而非个人编程。它包括支持程序规格说明、设计和演化的相关技术,而这些通常都与个人化的软件开发无关。为了帮助你全面理解软件工程的含义,图 1-1 对一些常见的问题及其回答进行了总结。

很多人将软件等同于计算机程序,其实这种理解是很狭隘的。在我们讨论软件工程时,软件不仅包括程序本身,而且还包括所有使程序能够正常使用的相关文档、库、支持网站、配置数据等。一个专业化开发的软件系统通常远不止一个程序。系统可能包含多个程序以及用于设置这些程序的配置文件。系统还可能包括描述系统结构的系统文档、解释如何使用该系统的用户文档,以及告知用户下载最新产品信息的 Web 网站。

这是专业软件开发与业余软件开发的一个重要区别。如果你只是自己编写一个程序,而且除你自己之外没有别的用户的话,你就不用费心编写程序指南和设计文档等。然而,如果你的软件有其他用户并且其他工程师会进行修改的话,你就必须在程序源代码之外提供其他附加信息。

软件工程师关心的是软件产品(即能卖给客户的软件)的开发。软件产品有以下两类。

问 题	答 案
什么是软件?	计算机程序和相关文档。软件产品可以针对特定客户开发,也可以面向一个通用的市场开发
好的软件具有哪些特性?	好的软件应当向用户提供所需的功能与性能,而且应当具备好的可维护性、可依赖性和可用性
什么是软件工程?	软件工程是一个工程学科,涵盖了软件生产的各个方面,从初始的构想到运行和维护
基本的软件工程活动有哪些?	软件规格说明、软件开发、软件确认和软件演化
软件工程和计算机科学有什么区别?	计算机科学关注理论和基础,而软件工程则关注开发和交付有用的软件的实践
软件工程和系统工程有什么区别?	系统工程关注基于计算机的系统开发的所有方面,包括硬件、软件和过程工程。软件工程是这个更加泛化的过程的一部分
软件工程面临的关键挑战是什么?	应对不断增长的多样性、缩短交付时间以及开发可信软件的要求
软件工程的成本有哪些?	软件开发成本约占总成本的 60%,测试成本占 40%。对于定制化软件而言,演化成本经常超过开发成本
最好的软件工程技术和方法是什么?	虽然所有的软件项目都必须进行专业化的管理和开发,但适合于不同类型的系统的技术各不相同。例如,游戏开发总是需要使用一系列的原型,而安全关键的控制系统开发则要求开发一个完整并且可分析的规格说明。没有任何方法和技术适用于所有系统
互联网给软件工程带来了哪些不同?	互联网不仅带来了大规模、高度分布式、基于服务的系统的开发,而且在互联网的支持下创造了改变软件经济模式的移动 App 产业

图 1-1 软件工程的常见问题

1. 通用软件产品,由软件开发组织开发,在市场上公开销售,可以独立使用。这类软件产品包括移动应用、数据库软件、字处理软件、绘图软件以及项目管理工具等。还包括用于特定目的的所谓的“垂直”应用产品,如图书馆信息系统、财务系统等。

2. 定制化软件产品,受特定的客户委托,由软件承包商专门为这类客户设计和实现。这类软件包括电子设备的控制系统、特定的业务处理系统、空中交通管制系统等。

这两类产品的一个关键区别在于:在通用软件产品中,软件规格说明由开发者自己确定,这意味着如果在开发过程中遇到问题,那么开发者可以重新思考所要开发的东西;而定制化软件产品的软件规格说明通常是由客户给出的,开发者必须按客户要求开发。

然而,这两类产品之间的界限正在变得越来越模糊。现在很多公司都会在一个通用软件产品基础上进行定制以满足特定客户的具体要求。ERP(企业资源规划)系统(例如,来自 SAP 和 Oracle 的系统)就是这种方法最好的例子。这种大规模的复杂系统通过加入所需要的业务规则、过程、报表等信息来适应不同企业的需要。

软件除了提供相应的功能外,作为一种产品还有一系列相关的质量属性。这些属性不直接涉及软件的功能,而是反映软件在执行时的行为以及源程序的结构、组织及相关的文档。软件对用户查询的响应时间以及程序代码的可理解性就属于这类属性(有时称为非功能性属性)。

对于一个软件系统,你所期望的特定属性集合显然取决于其应用方式。例如,飞机控制系统必须安全,交互式游戏必须响应快,电话交换系统必须可靠等。这些属性的泛化总结如图 1-2 所示,它们也是专业化软件系统的基本特性。

产品特性	描 述
可接受性	软件对于目标类型的用户而言必须是可接受的。这意味着软件必须可理解、有用,并且与用户使用的其他系统相兼容
可依赖性和信息安全性	软件可依赖性包括一系列特性,如可靠性、信息安全性、安全性。可依赖的软件即使在系统失效时也不应当导致物理或经济上的破坏。软件必须保证信息安全,使得恶意用户无法访问或破坏系统
效率	软件不应当浪费系统资源,例如存储和处理器周期。因此,效率包括响应性、处理时间、资源利用情况等
可维护性	软件应当能够通过演化满足客户变化的需求。这是一个关键属性,因为软件变更是一个变化的业务环境不可避免的要求

图 1-2 好的软件的基本属性

1.1.1 软件工程

软件工程是一门工程学科,涉及软件生产的各个方面,从最初的系统规格说明直到投入使用后的系统维护,都属于其学科范畴。在这个软件工程的定义中有以下两个关键词。

1. 工程学科。工程师让事物运转起来。他们在适当的地方应用理论、方法和工具。然而他们是选择性地使用这些理论、方法和工具,并且即使没有可用的理论和方法,他们也总是试图寻求问题的解决方案。工程师还认识到他们必须在组织和经济约束下工作,并且必须在这些约束之下寻求解决方案。

2. 软件生产的各个方面。软件工程不仅仅关注软件开发的技术过程,它也包括其他一些活动,例如软件项目管理以及支持软件开发的工具、方法和理论的开发。

工程都是要求在进度和预算范围内获得所要求的品质的成果。这经常涉及权衡决策:工程师不能是完美主义者。而为自己写程序的人则可以在程序开发上想花多少时间就花多少时间。

一般而言,软件工程师都会在其工作中运用系统化、有组织的方法,因为这通常是开发高质量软件最有效的方式。然而,工程都是为各种情况选择最恰当的办法,因而一个更有创造性却不那么正式的开发方法对于某些软件可能是恰当的选择。一种更加灵活的、支持快速变化的软件过程尤其适合于交互式的基于 Web 的系统或移动应用的开发,这类软件要求综合软件和图形化开发技能。

软件工程之所以重要有两方面的原因:

1. 个人和社会正越来越多地依赖于先进的软件系统。这就要求我们能够以经济而且快速的方式开发出可靠、可信的系统。

2. 从长远来看,运用软件工程方法和技术开发专业化的软件系统,比单纯作为个人编程项目编写程序更加便宜。无法有效应用软件工程方法将会导致更高的测试、质量保障和长期维护的成本。

软件工程中所使用的系统化方法有时被称为“软件过程”。软件过程是指实现软件产品开发的序列。所有的软件过程都包含以下 4 项基本活动:

1. 软件规格说明,其中客户和工程师定义所要开发的软件以及对其运行的约束;
2. 软件开发,对软件进行设计和编程实现;
3. 软件确认,对软件进行检查以确保它是客户所需要的;
4. 软件演化,对软件进行修改以反映客户和市场需求的变化。

如第2章中将解释的那样,不同类型的系统需要不同的软件开发过程。例如,飞机上所使用的实时软件必须在开发开始之前确定完整的规格说明。而在电子商务系统中,规格说明和程序通常是一起开发的。因此,根据所开发的软件类型的不同,这些通用的开发活动可以以不同的方式进行组织,并且以不同的详细程度进行描述。

软件工程与计算机科学和系统工程都相关。

1. 计算机科学关注支撑计算机和软件系统的基础理论和方法,而软件工程则关注软件开发过程中的实际问题。正如电子工程师必须具有一定的物理学知识一样,软件工程师必须具有一定的计算机科学知识。然而,计算机科学理论通常更适用于相对较小的程序。优雅的计算机科学理论很少与需要软件解决方案的大型复杂问题相关。

2. 系统工程关注复杂系统的开发和演化的各个方面,在这类系统中软件扮演着重要的角色。因而,系统工程关注硬件开发、政策和过程设计、系统部署,以及软件工程。系统工程师参与制定系统规格说明、定义系统总体体系结构,然后集成不同的部分以创建整个系统。

如在1.1.2节的讨论,有许多不同类型的软件。没有放之四海而皆准的软件工程方法和技术。然而以下4个相关的问题对许多不同类型的软件都有影响。

1. 异构性。当今越来越多的系统都是跨网络运行的分布式系统,其中包含不同类型的计算机和移动设备。除了在通用计算机上运行之外,软件也有可能要在移动电话和平板电脑上运行。有时还必须将新开发的软件与用不同的编程语言开发的老的遗留系统相集成。由此带来的挑战是,如何开发相应的技术以构造可靠且能够灵活应对这种异构性的软件。

2. 企业和社会的变革。随着新经济的发展和新技术的不断涌现,企业和社会正在发生着前所未有的快速变革。他们要能够快速地修改现有的软件同时开发新的软件。很多传统的软件工程技术都很耗时,新系统的交付往往远远滞后于计划。这些技术必须不断发展,从而使软件为其客户提供价值所需的时间能够缩短。

3. 信息安全与信任。软件与我们生活的各个方面都息息相关,因此软件能够取得人们的信任是很重要的。那些通过Web页面或Web服务接口访问的远程软件系统尤其如此。我们必须确保怀有恶意的人无法成功地攻击我们的软件,使信息安全能够得到保障。

4. 规模。软件开发必须能够在很大的范围内支持不同规模的系统,从移动或可穿戴设备中的很小的嵌入式系统到互联网上服务全球社区的基于云的系统。

为了应对这些挑战,我们需要新的工具和技术,且需要以创新性的方式结合并使用现有的软件工程方法。

1.1.2 软件工程的多样性

软件工程是生产软件的系统化的方法,它考虑了现实的成本、进度、可靠性等问题,以及软件客户和开发者的需要。所使用的特定的方法、工具和技术取决于开发软件的组织、软件的类型以及开发过程中所涉及的人。没有一个通用的软件工程方法和技术适合于所有的系统和公司。各种各样的软件工程方法和工具在过去的50多年里不断发展变化。然而,SEMAT倡议(Jacobson et al. 2013)认为存在一个基本的元过程,可以通过实例化创建不同种类的过程。这个元过程仍然处于早期发展阶段,可能是改进我们当前的软件工程方法的一个基础。

在考虑哪种软件工程方法和技术最重要时,也许最重要的因素是所开发的应用的类型。存在许多不同类型的应用。

1. 独立的应用。这是运行在个人计算机上的应用或者运行在移动设备上的应用。它们包含所有必要的功能,可以不用连接到网络上。这类应用的例子包括个人计算机上的办公软件、计算机辅助设计程序、图片处理软件、旅行应用、生产力应用等。

2. 基于事务的交互式应用。这类应用在远程计算机上运行,用户通过自己的计算机、手机或平板电脑进行访问。显然,这类应用包括 Web 应用,例如电子商务应用,这种应用可以让用户通过与一个远程系统交互来购买商品或服务。这类应用还包括业务系统,一个企业通过 Web 浏览器或者特殊的客户端程序以及基于云的服务让用户访问他们的系统,例如邮件和照片共享。交互式应用通常包含大规模的数据存储,这些数据在每一次事务中被访问和更新。

3. 嵌入式控制系统。这类应用有一个软件控制系统来控制和管理硬件设备。从数量上看,嵌入式系统远远多于任何其他类型的系统。嵌入式系统的例子包括移动电话中使用的软件、汽车上控制防抱死刹车的软件,以及微波炉上控制烹饪过程的软件。

4. 批处理系统。这是一类业务系统,被设计用来处理大批量的数据。它们处理大量的单个输入以创建相应的输出。这类系统包括定期计费系统,例如电话计费系统和工资支付系统。

5. 娱乐系统。这类系统主要是个人用户用于娱乐。大多数的这类系统是运行在专用的游戏机硬件上的各种游戏。这类系统所提供的交互质量是娱乐系统和其他系统的一个重要区别。

6. 建模和仿真系统。科学家和工程师开发这类系统来模拟物理过程或环境,其中包括很多独立且相互交互的对象。这些系统通常是计算密集型的,需要高性能的并行系统来运行。

7. 数据收集和分析系统。数据收集系统从环境中收集数据,并将数据发送到其他系统进行处理。这些软件可能要和传感器进行交互,并且经常安装在恶劣环境中(例如安装在发动机内部或者是野外)。“大数据”分析可以使用基于云的系统来进行统计分析,并寻找所收集的数据中的关系。

8. 系统之系统。这类系统在企业或其他大型组织中使用,由其他一些软件系统组成。其中一些系统是通用软件产品,例如 ERP 系统。其他的一些系统则可能是专门为这个环境编写的软件。

当然,这些不同软件类型之间的边界是模糊的。如果你开发一个手机游戏,必须与手机软件开发者一样考虑相同的约束(电量、硬件交互)。而批处理系统则经常和基于 Web 的事务系统一起使用。例如,一个公司的差旅费报销申请可能通过 Web 应用提交,但却需要批处理应用来处理以实现按月支付。

每种类型的系统需要专门的软件工程技术,因为软件有不同的特点。例如,一个汽车上的嵌入式控制系统的安全性十分重要,在车上安装时是烧录到 ROM(只读存储器)中的。因此,该软件的修改十分昂贵。这样的系统需要全面的验证和确认,以确保销售出去之后不得不召回以修复软件问题的可能性最小化。用户交互很少(或者也许根本不需要),因此不需要使用一个依赖于用户界面原型的开发过程。

对于交互式的基于 Web 的系统或移动应用,迭代式开发和交付是最好的方法,其中系统由可复用的构件组成。然而,这一方法对于系统之系统就不适用了,因为其中的系统交互的详细规格说明必须提前定义好,以使得每个系统都可以独立进行开发。

然而,还是有一些软件工程基础适用于所有类型的软件系统。

1. 软件系统开发过程应当是受管理的并且被开发人员所理解。软件开发组织应当对开发过程进行计划,并对要开发什么以及什么时候完成很清楚。当然,应当使用什么样的特定过程则取决于你所开发的软件的类型。

2. 可靠性和性能对所有类型的系统来说都很重要。软件应该按照所期待的方式运行,不会发生失效,并且在用户需要的时候是可用的。它在运行过程中应该确保安全,并且针对外部攻击尽可能保证信息安全。系统应当高效地运行且不会浪费资源。

3. 理解和管理系统规格说明和需求(软件应该做什么)是很重要的。你必须知道不同的客户和用户对于系统的期望是什么,然后你必须管理他们的期望以便能够在预算范围内按时交付一个有用的系统。

4. 你应该尽可能高效地使用已有的资源。这就意味着,你应该在一些适当的地方复用已开发的软件,而不是重新写一个新软件。

关于开发过程、可依赖性、需求、管理和复用的基本概念是本书的重要主题。不同的方法以不同的方式反映这些概念,但它们是所有专业化软件开发的基础。

这些基础与软件开发所使用的编程语言无关。本书不涉及特定的编程技术,因为不同类型的系统在编程技术方面差异很大。例如,动态语言(例如 Ruby)是交互式系统开发的一个正确选择,然而这类语言完全不适合嵌入式系统工程。

1.1.3 互联网软件工程

互联网和万维网的发展已经对我们的生活产生了深远的影响。最初,Web 只是一种可以被广泛访问的信息源,对软件系统几乎没有影响。那时的软件系统在本地计算机上运行,且只是由同一组织内部的人访问。2000 年左右,Web 开始演化,浏览器添加了越来越多的功能。这就意味着基于 Web 的系统的开发允许用户通过浏览器而非特别开发的用户界面来访问这些系统。这就导致了大量通过 Web 访问、提供创新性服务的新系统产品的开发。这些系统通常依赖广告赞助,广告显示在用户屏幕上,不需要用户直接付费。

除了这些系统产品,Web 浏览器的发展使得浏览器可以运行小程序并且进行一些本地处理,这导致了业务和组织软件的变化。不同于将写好的软件部署在用户的个人计算机上运行,这些软件部署在 Web 服务器上。这就使得修改和升级软件变得更加便宜,因为不需要在每台个人计算机上安装软件。这也降低了成本,因为用户界面开发非常昂贵。因此,只要有可能,公司就会选择将软件系统移动到基于 Web 的交互模式上。

软件即服务的思想(第 17 章)在 21 世纪初被提出来。这一思想现在已经成为基于 Web 的系统产品(例如 Google 移动应用、微软的 Office 365、Adobe 的 Creative 套件)交付的标准方法。越来越多的软件运行在远端的“云”上而非本地的服务器上,并通过互联网访问。计算云是由大量相互连接并由许多用户共享的计算机系统组成的。用户并不购买软件而是按照软件的使用量进行付费,或者让用户免费访问并通过让用户观看在屏幕上展示的广告来获得回报。如果你使用基于 Web 的邮件、存储或视频这样的服务,那么你正在使用基于云的系统。

Web 的出现使业务软件的组织方式发生了剧烈的改变。在没有 Web 之前,绝大多数的业务应用都是巨石应用,单个的程序运行在单个计算机或者计算机集群上,只有组织内部的本地通信。现在,软件是高度分布式的,有时会跨越整个世界。业务应用不再是从头开始编写程序,而是包含了对构件和程序的大规模复用。

软件组织中的这一变化已经对基于 Web 的系统的软件工程造成了巨大的影响。例如：

1. 软件复用已经成为构建基于 Web 的系统的主流方法。当你构造这样的系统时需要考虑如何在已存在的软件构件和系统基础上装配系统，这些构件和系统经常捆绑在一个框架中。

2. 人们普遍承认提前确定这些系统的所有需求是不切实际的。基于 Web 的系统总是增量开发和交付的。

3. 软件可以使用面向服务的软件工程来实现，其中软件构件是独立的 Web 服务。本书将在第 18 章中介绍这种软件工程方法。

4. AJAX (Holdener 2008) 和 HTML5 (Freeman 2011) 等界面开发技术已经出现，这些技术支持 Web 浏览器中的富客户端界面的创建。

与其他类型的软件一样，前面讨论的软件工程的基本思想同样适用于基于 Web 的软件。基于 Web 的系统正在变得越来越大，因此应对规模和复杂性的软件工程技术与这些系统相关。

1.2 软件工程职业道德

和其他工程学科一样，软件工程是在一个社会和法律框架中进行的，这个框架限制了在这个领域中工作的人们的自由。作为一个软件工程师，你必须接受你的工作包含更广阔的责任而不仅仅是技术能力的应用。如果你想作为一个专业工程师得到尊重，那么你的行为必须合乎职业道德并且有责任感。

软件工程师必须坚持诚实正直的行为准则，这是不言而喻的。他们不能用所掌握的知识 and 技能做不诚实的事情，更不能给软件工程行业抹黑。然而，在有些方面，某些行为没有法律加以规范，只能靠职业道德来约束，这种约束是软弱无力的。其中包括：

1. 保密。通常你必须严格保守雇主或客户的机密，无论你们是否签署了保密协议。

2. 工作能力。你不应该对你的能力水平进行虚假陈述。你不应该故意接受超出自己能力范围的工作。

3. 知识产权。你应当知晓有关专利权、著作权等知识产权的地方法律，必须谨慎行事，确保雇主和客户的知识产权受到保护。

4. 计算机滥用。你不应该运用自己的技能滥用他人的计算机。滥用计算机有时对他人影响不大（如在雇主的机器上玩游戏），但有时后果非常严重（传播病毒）。

在这个方面职业协会和机构肩负重任。ACM、IEEE 和英国计算机协会等组织颁布了职业行为准则或职业道德准则，凡是加入这些组织的成员必须严格遵守。这些行为准则只涉及基本的道德行为。

ACM 和 IEEE 还联合推出了一个关于职业道德和职业行为的准则，有两个版本：一个比较简短，见图 1-3；另一个版本 (Gotterbarn, Miller, and Rogerson 1999) 较长，增加了一些细节和要义。这个准则背后的出发点在完整版的前两段中进行了总结。

计算机在商业、工业、政府、医药、教育、娱乐和整个社会中的核心作用日渐突出。软件工程师是直接参与或讲授软件系统的分析、规格说明、设计、开发、认证、维护和测试的人员。基于在软件系统开发中的地位，软件工程师可能将事情做好也可能做坏，还可能让他人或影响他人将事情做好或做坏。为了最大限度地保证自己的工作是有意义的，软件工程师必须保证使软件工程业成为对社会有益的、受人尊敬的行业。基于以上保证，软件工程师应当

遵守下面的道德和职业行为准则^①。

本行为准则包括 8 项基本原则，针对包括软件工程行业的从业者、教育者、管理者、监督者、政策制定者、接受培训者和学生在内的职业软件工程师。这 8 条原则阐明了个人、团队和机构之间职业道德上的责任关系以及他们在其中应该履行的基本义务。每一原则的条款都表述了这些关系中的一些义务。这些义务既基于软件工程师的人性，也对那些受他们的工作和软件工程实践的独特环境影响的人们表示出特别的关怀。本准则把这些内容规定在任何一个自称或渴望成为软件工程师的义务中。

软件工程职业道德和行为准则

(ACM/IEEE-CS 软件工程职业道德和行为规范联合工作组)

前言

准则的简要版对其中的愿望做了高度抽象的概括；完整版中的条款对这些愿望进行了细化，并给出了实例，用以规范软件工程专业人员的工作方式。没有这些愿望，所有的细节都会变得教条而又枯燥；而没有这些细节，愿望就会变得高调而空洞。只有将二者紧密结合才能形成有机的行为准则。

软件工程师应当做出承诺，使软件的分析、规格说明、设计、开发、测试和维护等工作对社会有益且受人尊重。基于对公众健康、安全和福利的考虑，软件工程师应当遵守以下 8 条原则：

1. 公众感。软件工程师应始终与公众利益保持一致。
2. 客户和雇主。软件工程师应当在与公众利益保持一致的前提下，保证客户和雇主的利益最大化。
3. 产品。软件工程师应当保证他们的产品以及相关的修改尽可能满足最高的行业标准。
4. 判断力。软件工程师应当具备公正和独立的职业判断力。
5. 管理。软件工程师管理者和领导者应当维护并倡导合乎道德的有关软件开发和维护的管理方法。
6. 职业感。软件工程师应当弘扬职业正义感和荣誉感，尊重社会公众利益。
7. 同事。软件工程师应当公平地对待和协助每一位同事。
8. 自己。软件工程师应当毕生学习专业知识，倡导合乎职业道德的职业活动方式。

图 1-3 ACM/IEEE 道德准则 (ACM/IEEE-CS 软件工程职业道德和行为规范联合工作组，

简要版：<http://www.acm.org/about/se-code>)

(© 1999 ACM、IEEE 版权所有)

有些时候不同的人会有不同的观点和目标，这时你很有可能会面对道德困境。例如，如果你原则上不赞成公司高级管理层的决策，该怎么办？显然，这取决于所涉及的人以及不赞成的原因。是在团队内部坚持自己的观点并据理力争，还是坚持原则毅然辞职？如果你觉得一个软件项目有问题，你会选择在什么时候向管理层报告呢？如果只是在怀疑，这时向管理层报告未免有点过于敏感；如果拖了很久才报告，则有可能导致难题无法解决。

在我们的职业生涯中，每个人都会面临这种困境。幸运的是，在大多数情况下，这些困境要么不那么严重，要么不难解决。如果困境无法解决，那么工程师会面对另一个问题。有操守的做法是辞职，但这样做会影响到其他人，例如配偶或孩子。

当雇主的行为违背道德时，职业工程师的处境会很困难。例如，一个公司负责开发一个安全关键性系统，由于时间太紧而篡改了安全的确认记录。这时工程师是应该保守秘密，还是提醒客户注意，还是以某种方式揭露所交付的系统可能不安全？

这里的问题在于安全不是绝对的。虽然系统没有按照预定义的准则进行确认，但这些

① ACM/IEEE-CS 软件工程职业道德和行为规范联合工作组，简要版前言。<http://www.acm.org/about/se-code>，© 1999 国际计算机学会 (Association for Computing Machinery, ACM) 以及电气和电子工程师协会 (Institute of Electrical and Electronics Engineers, IEEE) 版权所有。

准则可能过于苛刻。系统实际上可能会在整个生命周期中一直保持安全运行。还有一种可能性,即使进行了适当的确认,最终系统仍然发生失效并导致事故。在早期披露这些问题将使雇主和其他雇员蒙受损失,如果隐瞒这些问题则又会对其其他人不利。

在这个问题上必须有自己的主见。这里恰当的道德立场取决于所涉及的人的观点。潜在的灾难、灾难的严重程度以及灾难的受害者都将影响决策。如果情况非常危险,就应该通过一定的方式披露出来,但同时还应该尊重雇主的权利。

另一个道德问题是参与军事项目和核项目。许多人已强烈地意识到这个问题,不想参与到军事项目中。有的人则愿意参与军事系统开发,但不愿意参与武器系统开发。但还有一些人感觉国家安全是压倒一切的原则,对于参与武器系统开发没有道德上的反感。

在上述情况中,雇主和所有雇员事先相互沟通各自的观点非常重要。若以一个组织的形式参与军事系统或核系统的开发,应该确保每个雇员都自愿接受工作安排。同样,如果雇员已经明确表示他们不愿意参与这类项目,雇主也不应该在日后给他们施加压力。

随着软件密集型系统越来越多地渗透到人们的日常工作和生活中,一般所说的道德和职业责任问题受到了越来越多的关注。可以从哲学的角度研究道德问题的基本原理,而软件工程职业道德的研究可以参照这些基本原理。这是 Laudon (Laudon 1995) 和 Johnson (Johnson 2001) 所采取的方法。更近的一些著作(例如 Tavani 2013)则引入了信息世界伦理道德的思想,同时覆盖了哲学背景以及实践和法律问题。其中包含了面向技术用户以及开发人员的道德问题。

我认为用哲学的研究方法太过抽象,很难与我们的日常生活联系起来。因此,我推崇职业行为准则(Bott 2005; Duquenoy 2007)中更加具体的方法。道德问题的研究最好要联系软件工程的实际,而不是将其作为一个孤立的问题。因此,我没有以一种抽象的方式讨论软件工程道德,而是在练习中提供了一些例子作为小组讨论的基础。

1.3 案例研究

为了更清楚地阐述相关的软件工程概念,本书中用了4种不同类型的系统作为例子。本书有意识地避免使用单一的案例研究,主要是因为本书要传递的一个关键信息就是软件工程的实践取决于所要开发的系统类型。因此,在讨论相关的概念(例如安全性、可依赖性、系统建模、复用等)时会有针对性地选择合适的例子。

本书用作案例的系统类型如下。

1. 嵌入式系统。这类系统中的软件控制硬件设备并嵌入该设备中。嵌入式系统的典型问题包括物理尺寸、响应性、电量管理等。本书嵌入式系统所用的例子是一个面向糖尿病患者的胰岛素泵控制软件系统。

2. 信息系统。这类系统的主要目的是管理和提供对信息数据库的访问服务。信息系统的主要问题包括信息安全、可用性、隐私以及保持数据的完整性。本书信息系统所用的例子是一个医疗记录系统。

3. 基于传感器的数据采集系统。这类系统的主要目的是从多个传感器收集数据,并以适当的方式处理数据。这类系统的关键需求是可靠性(甚至是在极端环境条件下)以及可维护性。本书数据采集系统所用的例子是一个野外气象站。

4. 支持环境。这类系统集成了一系列软件工具来支持某种活动。Eclipse (Vogel 2012) 这样的编程环境可能是本书读者最熟悉的一种环境。本书中介绍了一个用于支持学生校内学

习的数字化学习环境。

本章将逐一对这些系统进行介绍,关于这些系统更多的信息可以从本书的网站 (software-engineering-book.com) 上获得。

1.3.1 胰岛素泵控制系统

胰岛素泵是一个模拟胰腺(一种人体组织)运转的医疗系统。该系统的软件控制部分是一个嵌入式系统,它从传感器收集数据,然后控制泵输送指定剂量的胰岛素给患者。

糖尿病患者使用这个系统。糖尿病是一种常见病症,是由于人体无法产生足够数量的胰岛素而引起的。胰岛素在血液中起到促进葡萄糖代谢的作用。糖尿病的传统治疗方法是长期规律地注射人工胰岛素。糖尿病患者使用一种外部仪器定期测量自己的血糖值,然后计算所需要注射的胰岛素剂量。

这种治疗方法的问题在于,血液中的胰岛素浓度不仅与血液中的葡萄糖浓度相关,还与最后一次注射胰岛素的时间有关。这种不规律的检查有可能导致血糖浓度太低(当胰岛素太多时)或血糖浓度太高(当胰岛素太少时)。短时间内的低血糖是一种比较严重的情况,会导致暂时的脑功能障碍,最后失去知觉甚至死亡。长期处于高血糖则会导致眼睛损伤、肾损伤和心脏问题。

当前在微型传感器发展方面取得的进步使得自动胰岛素输送系统开发成为可能。这个系统监控血糖浓度,根据需要输送适当的胰岛素。类似这样的胰岛素输送系统现在已经有了,感觉很难控制自己的胰岛素水平的病人已经在使用这类系统了。将来还有可能将这样的系统永久地植入糖尿病患者体内。

该系统使用一个植入人体内的微传感器来测量一些血液参数,这些参数与血糖浓度成正比。这些参数被送到泵控制器。控制器计算血糖浓度,得出胰岛素需要量,然后向一个小型化的泵发送信号使之通过持久连接的针头输送胰岛素。

图 1-4 给出了胰岛素泵的硬件构件和组成结构。要理解本书中的这个例子,你所要知道的就是血液传感器测量血液在不同情况下的电传导率,而这些值是和血糖浓度相关的。控制器发送一个脉冲信号,胰岛素泵就会输送一个单位的胰岛素。因此输送 10 个单位的胰岛素,控制器就会向泵发送 10 个脉冲信号。图 1-5 是一个 UML (统一建模语言) 活动模型,描述了如何将输入的血糖浓度值转换为驱动胰岛素泵的命令序列。

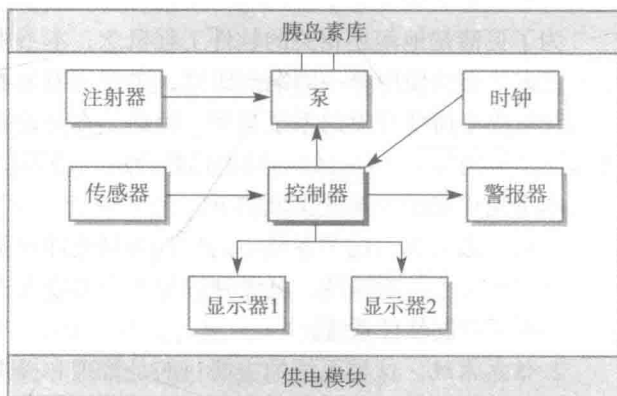


图 1-4 胰岛素泵的硬件体系结构

显然,这是一个安全攸关的系统。如果泵无法运行或者运行出问题,那么将会危及病人的健康,甚至使得病人因血糖浓度过低或者过高而引发昏迷或损伤脏器。因此,该系统必须满足以下这两个关键性的高层需求:

1. 当需要时这个系统应当能够输送胰岛素;
2. 系统应当可靠地运行,并根据当前血糖浓度输入正确剂量的胰岛素。

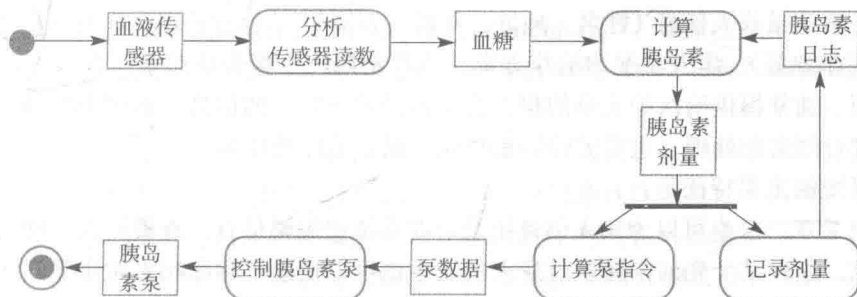


图 1-5 胰岛素泵的活动模型

因此，系统的设计和实现必须确保能满足这些需求。本书将在后面的章节介绍更多的具体需求，并讨论如何确保系统的安全性。

1.3.2 心理健康治疗病人信息系统

支持心理健康治疗的病人信息系统（Mentcare 系统）是一个医疗信息系统，维护着遭遇心理健康问题的病人信息以及他们所接受的治疗信息。大多数有心理问题的人不需要住院治疗，但是需要定期去那些了解他们详细病情的专科诊所看医生。为了方便病人看病，这些诊所不只是开在医院里，也有可能开在当地私人诊所或社区中心。

Mentcare 系统（图 1-6）是将在诊所使用的一个病人信息系统。它使用了一个集中式的病人信息数据库，但也可以在笔记本电脑上使用，这使得该系统可以在没有安全的网络连接的地方访问和使用。在有安全的网络连接时，他们使用数据库中的病人信息，但是可以下载病人记录的本地拷贝并在没有网络连接时使用。这个系统不是一个完备的医疗记录系统，因此没有记录其他的医疗信息。然而，它可以与其他诊所信息系统交互并交换数据。

该系统有以下两个目的：

1. 生成管理信息，使健康服务部门能够据此评估本地和政府目标的实现情况；

2. 及时为医护人员提供相关信息以支持对病人的治疗。

有心理健康问题的病人有时候会失去理性并有些紊乱，因此可能会错过预约时间、故意或意外遗失处方和药品、不听医生嘱咐、对医护人员提出无理要求、不期而至等。在少数情况下，他们还可能对自己和他人造成危害。他们可能经常更换住址，有时会长期或短期地离家出走。如果病人具有危险性，那么他们可能需要被“隔离”，即限定在一个安全的医院接受治疗 and 观察。

该系统的用户包括诊所工作人员，例如医生、护士和健康随访员（上门检查治疗状况的护士）。非医务用户包括负责预约的接待员、负责维护系统医疗数据的工作人员，以及负责生成报告的行政人员。

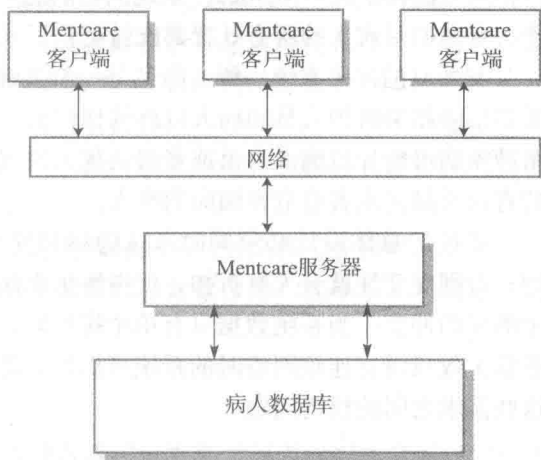


图 1-6 系统的组织结构

系统负责记录病人信息(姓名、地址、年龄、亲属等)、诊疗情况(日期、责任医生、对病人的主观印象等)、病人状况和治疗方案。系统定期产生报告供医务人员和卫生部门管理者使用。通常提供给医护人员的报告关注的是单个病人的信息,而用于提供给管理者的报告则会进行匿名化处理,主要关注的是总体状况、治疗费用等。

这个系统的主要特征是:

1. 病例管理。医生可以为病人创建记录、在系统中编辑信息、查看病人历史等。系统支持数据汇总,这样某个先前未接触过病人的医生也可以快速了解此病人的主要问题和当前的治疗情况。

2. 病人监测。系统定期监测那些正在接受治疗的病人的记录,若发现可疑问题就会发出提醒。因此,若某个病人很长时间都没有看医生了,系统就会发出通知。此监测系统最重要的一个特点是,能够对强制隔离的病人保持跟踪,以确保在正确的时间能够对其进行合乎法律要求的例行检查。

3. 管理报告。系统产生月度管理报告,显示每个诊所接治的病人数目、进入和离开护理系统的病人数目、采取强制隔离的病人数目、处方药物的使用情况及其价格等。

有两项法律条款影响该系统。一个是有关个人信息保密性的数据保护法,另一个是有关强制监禁被认为对病人自己和其他人会造成伤害的病人的心理健康法。心理健康在这方面是很独特的,因为它是唯一可以违背病人的意愿、建议对其进行看管的医学专业。这是受到非常严格的立法保护的。Mentcare 系统的目的之一就是确保工作人员总是能够按照法律的规定工作,且他们对病人的所有处置都能被记录下来并在必要的时候用于司法审查。

与所有的医疗系统一样,隐私是一个关键性的系统需求。病人信息是严格保密的,不能暴露给除相关医护人员和病人以外的任何人。Mentcare 系统也是一个安全攸关的系统。一些精神疾病可能导致病人自杀或者对其他人造成人身伤害。系统应尽可能向医护人员警示潜在的有自杀倾向或者有危害倾向的病人。

系统的总体设计必须同时考虑隐私和安全两个方面的需求。系统必须在需要时是可用的;否则安全性就会大打折扣,使得医生无法及时为病人拿出正确的治疗方案。这里存在一个潜在的冲突。当系统数据只有单个拷贝时,隐私保护是最容易做到的。然而,为确保在服务器失效或没有连接网络时的系统可用性,需要维护多份数据拷贝。本书后续章节将会讨论这些需求之间的权衡问题。

1.3.3 野外气象站

为了监控偏远地区的气候变化、提高气象预报的准确度,那些幅员辽阔的国家的政府会选择在偏远地区部署几百个气象站。这些气象站通过一组装置来采集气象数据,比如温度、气压、光照、降雨、风速和风向。

野外气象站只是一个更大的系统的一部分(见图 1-7),该系统是一个从气象站采集数据并将其提供给其他系统处理的气象信息系统。图 1-7 中的系统包括:

1. 气象站系统(Weather station)。该系统负责收集气象数据,做一些初始处理,然后传输给数据管理系统。

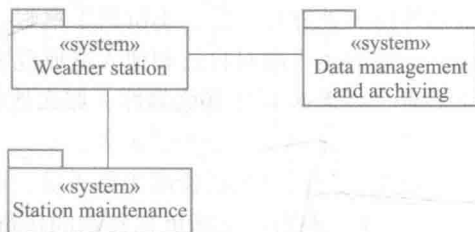


图 1-7 气象站的上下文环境

2. 数据管理与存档系统 (Data management and archiving)。该系统从所有的野外气象站收集数据, 进行数据处理与分析, 将数据存储为容易被其他系统 (如天气预报系统) 检索的格式。

3. 气象站维护系统 (Station maintenance)。该系统可以通过卫星与所有野外气象站通信, 监控它们的运行状态, 并报告出现的问题。还可以更新这些气象站上的嵌入式软件。当某个野外气象站系统出问题, 该系统还可以用来远程控制气象站系统。

图 1-7 中使用 UML 包的符号来表示每个系统都是一个构件的集合, 并且采用 UML 构造型 (stereotype) «system» 表示所有的独立系统。包之间的关联表示包与包之间存在信息交换, 但目前我们没必要对它们做更详细的描述。

每一个气象站都有许多采集各种天气数据的仪器, 比如风速、风向、气温、气压、24 小时降雨量等。所有这些设备都是在软件系统的控制下周期性地读入并管理所采集到的数据。

气象站系统进行气象数据观察的频率是很高的, 例如对温度的测量需每分钟进行一次。然而, 由于卫星带宽相对较窄, 气象站系统需要在本地对数据进行一些处理和存储。当数据收集系统请求数据时, 气象站系统提交存储在本地的数据。如果通信连接不成功, 气象站系统在本地继续保留数据, 直到通信恢复。

每个气象站都是由独立电池供电的, 而且完全是自我管理的, 没有外部供电或有缆电缆存在。所有的通信都是通过速度相对较慢的卫星连接的。气象站必须包含自我充电的机制, 例如太阳能或风能。由于它们是部署在野外的, 直接暴露在各种恶劣环境条件下, 还有可能被动物毁坏。因此气象站软件不能仅仅进行数据采集, 还必须做到以下几点:

1. 监控仪器、电源、通信硬件, 并向管理系统报告故障。
2. 管理系统电源, 确保电池在环境条件允许的情况下能够充电, 也确保在恶劣天气情况 (例如大风天气) 下及时关闭发电机以免受到破坏。
3. 允许动态配置, 在部分软件版本更新时或者当系统失效而切换备份装置时。

气象站必须是自我管理和无人看管的。这就意味着尽管数据采集功能本身相对简单, 但所安装的软件是复杂的。

1.3.4 学校数字化学习环境

许多教师都认为使用交互式软件系统来支持教育既可以提高学习者的兴趣又可以使得学生更深刻地理解知识。然而, 对于什么是“最好的”计算机支持的学习策略并没有广泛共识。在实践中, 教师通常会使用多种不同的交互式、基于 Web 的工具来支持学习。所用的工具取决于学习者的年龄、他们的文化背景、他们的计算机使用经验、可用的设备以及相关教师的偏好等。

数字化学习环境是一个框架, 其中包含一些通用工具以及一些针对特殊目的设计的工具, 外加一组满足使用该系统的用户需求的应用。该框架提供了身份认证服务、同步和异步通信服务以及存储服务 etc 通用服务。

该环境的每一个版本中所包含的工具由教师和学习者根据他们自己的需要来选择。这些工具可以是电子表格等通用的应用, 也可以是管理作业提交和评分、游戏和模拟的虚拟学习环境 (Virtual Learning Environment, VLE) 等学习管理应用。它们还可以包括一些特定的内容, 例如关于美国内战的内容以及观看和标注内容的应用。

图 1-8 是一个面向 3 ~ 18 岁学生在校内使用的数字化学习环境 (iLearn) 的高层体系结构模型。这里采用了分布式系统设计, 其中该学习环境所有的构件都是可以从互联网上任何地方访问的服务。不要求所有的学习工具汇集在一个地方。

该系统是一个面向服务的系统, 其中所有的系统构件都被认为是一个可替换的服务。该系统中有以下 3 种类型的服务。

1. 公共服务。提供与应用无关的基础功能, 可以被系统中的其他服务所使用。公共服务通常是特别针对该系统开发或修改的。

2. 应用服务。提供特定的应用 (例如邮件、会议、图片共享等) 以及针对特定教育内容 (例如科学电影或历史资源) 的访问手段。应用服务是为该系统特别购买的或从互联网上免费获取的外部服务。

3. 配置服务。用于使用特定的应用服务集合对学习环境进行调整和设置, 并定义如何在学生、教师以及学生父母之间共享服务。

该学习环境的设计使得服务可以在有了新服务之后进行替换, 从而可以提供该系统的不同版本以适应不同年龄的用户的需要。这意味着该系统必须支持以下两个层次上的服务集成。

1. 集成服务, 提供应用编程接口 (Application Programming Interface, API), 其他服务可以通过 API 进行访问。这样一来, 服务到服务的直接通信就成为可能。身份认证服务是一个集成服务的例子。其他服务可以调用身份认证服务来对用户进行认证, 而不是使用自己的认证机制。如果用户已经通过认证, 身份认证服务可以通过 API 将认证信息直接发送给另一个服务, 用户不需要重新对自己进行认证。

2. 独立服务, 其运行与其他服务无关, 可以直接通过浏览器界面进行访问。可以通过显式的用户动作 (例如复制粘贴) 与其他服务共享信息。每个独立服务都可能会要求重新进行身份认证。

如果一个独立服务得到了广泛使用, 开发团队可以集成该服务, 从而使之成为一个集成的和支持性的服务。



图 1-8 数字化学习环境 (iLearn) 的体系结构

要点

- 软件工程是一门覆盖软件生产的各个方面的工程学科。
- 软件不仅是程序, 还包括系统用户、质量保证人员以及开发者所需要的所有电子文档。软件产品的基本属性是可维护性、可依赖性、信息安全性、效率以及可接受性。
- 软件过程包括软件开发过程中所涉及的所有活动。软件规格说明、开发、确认和演化这些高层活动是所有软件过程中的一部分。
- 世界上存在着很多不同类型的系统。每一种类型的系统的开发都需要一种与之相适

应的软件工程工具和技术。几乎不存在适用于所有类型的系统的软件设计和实现技术。

- 软件工程的基本思想适用于所有的软件系统。这些基本思想包括受管理的软件过程、软件的可依赖性和信息安全性、需求工程和软件复用。
- 软件工程师对软件工程行业和整个社会负有责任，不应该只关心技术问题，而应该对影响他们工作的道德问题有所知晓。
- 职业协会发布的行为准则定义了道德和职业标准。这些准则为协会成员设定了所期望的行为标准。

阅读推荐

《Software Engineering Code of Ethics Is Approved》这篇文章介绍了制定 ACM/IEEE 职业道德准则的背景，该准则包括一个简要版和一个较长的版本。(Comm. ACM, D. Gotterbarn, K. Miller, and S. Rogerson, October 1999) <http://dx.doi.org/10.1109/MC.1999.796142>

《A View of 20th and 21st Century Software Engineering》这本书是第一代最杰出的软件工程师对软件工程的回顾和展望。Barry Boehm 指出了一些永恒的软件工程原则，但也认为一些广泛使用的实践已经过时了。(B. Boehm, Proc. 28th Software Engineering Conf., Shanghai, 2006) <http://dx.doi.org/10.1145/1134285.1134288>

《Software Engineering Ethics》是《IEEE Computer》的一个专刊，其中有一些关于此问题的文章。(IEEE Computer, 42 (6), June 2009)

《Ethics for the Information Age》是一本内容很丰富的书，其中覆盖了信息技术 (IT) 道德的所有方面，不仅是针对软件工程师的道德要求。这是正确的方法，因为你真的需要在一个更广阔的道德框架下去理解软件工程道德。(M. J. Quinn, 2013, Addison-Wesley)

《The Essence of Software Engineering: Applying the SEMAT kernel》这本书讨论了建立一种支撑所有的软件工程方法的通用框架的思想。该框架可以通过适应性修改被用于各种不同类型的系统和组织。我个人对这样一种通用的方法在实践中是否具有现实性有所怀疑，但本书中有一些有趣的思想值得尝试。(I. Jacobsen, P-W Ng, P. E. McMahon, I. Spence, and S. Lidman, 2013, Addison-Wesley)

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap1/>

支持视频的链接: <http://software-engineering-book.com/videos/software-engineering/>

案例研究描述的链接: <http://software-engineering-book.com/case-studies/>

练习

- 1.1 为什么面向一个客户开发的专业化软件不仅仅包含所开发和交付的程序？
- 1.2 通用的软件产品开发和定制化软件开发之间最重要的区别是什么？这在实践中对于通用软件产品的用户意味着什么？
- 1.3 软件产品应该具有的 4 个重要属性是什么？另外举出 4 个可能有意义的属性。
- 1.4 除了异构性、企业和社会的变革、可信和信息安全之外，说一说软件工程在 21 世纪有可能面对的其他问题和挑战（提示：想一想环境）。
- 1.5 参考 1.1.2 节讨论的应用类型，举例说明为什么设计和开发不同类型的应用需要特殊化

的软件工程技术。

- 1.6 解释为什么过程、可依赖性、需求管理、复用这些基本的软件工程原则与所有类型的软件系统都相关。
- 1.7 解释 Web 的普遍使用是如何改变软件系统和软件系统工程的。
- 1.8 讨论一下职业工程师是否应该和医生或律师一样颁发资格证书。
- 1.9 对图 1-4 中的 ACM/IEEE 职业道德准则中的每一条举出一个恰当的例子加以说明。
- 1.10 为了反恐,很多国家正计划开发或正在开发一种对其大量公民及其行动跟踪的计算机系统。显然这是侵犯个人隐私权的做法。对开发此类系统的道德伦理问题进行讨论。

参考文献

Bott, F. 2005. *Professional Issues in Information Technology*. Swindon, UK: British Computer Society.

Duquenoy, P. 2007. *Ethical, Legal and Professional Issues in Computing*. London: Thomson Learning.

Freeman, A. 2011. *The Definitive Guide to HTML5*. New York: Apress.

Gotterbarn, D., K. Miller, and S. Rogerson. 1999. "Software Engineering Code of Ethics Is Approved." *Comm. ACM* 42 (10): 102–107. doi:10.1109/MC.1999.796142.

Holdener, A. T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Jacobson, I., P-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. 2013. *The Essence of Software Engineering*. Boston: Addison-Wesley.

Johnson, D. G. 2001. *Computer Ethics*. Englewood Cliffs, NJ: Prentice-Hall.

Laudon, K. 1995. "Ethical Concepts and Information Technology." *Comm. ACM* 38 (12): 33–39. doi:10.1145/219663.219677.

Naur, P., and Randell, B. 1969. Software Engineering: Report on a conference sponsored by the NATO Science Committee. Brussels. <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf>

Tavani, H. T. 2013. *Ethics and Technology: Controversies, Questions, and Strategies for Ethical Computing*, 4th ed. New York: John Wiley & Sons.

Vogel, L. 2012. *Eclipse 4 Application Development: The Complete Guide to Eclipse 4 RCP Development*. Sebastopol, CA: O'Reilly & Associates.

软件过程

目标

本章的目标是介绍软件过程（软件生产的一组相互连贯的活动）的思想。读完本章，你将：

- 理解软件过程和软件过程模型的概念；
- 了解3个通用的软件过程模型以及它们的适用情形；
- 了解需求工程、开发、测试和演化这几个基本的软件过程活动；
- 理解为什么软件过程要有效地组织以应对软件需求和设计上的变化；
- 理解软件过程改进的思想以及影响软件过程质量的因素。

软件过程是完成软件产品生产的一组相互关联的活动。如第1章中所述，有许多不同类型的软件系统，没有放之四海而皆准、适用于所有类型系统的软件工程方法。因此，也没有放之四海而皆准的软件过程。不同企业中所使用的过程取决于所开发的软件的类型、软件客户的需求以及开发软件的人的技能。

虽然有许多不同的软件过程，但它们都必须以某种形式包含在本书第1章所介绍的4个最基本的软件工程活动中：

1. 软件规格说明。软件的功能以及对于软件运行的约束必须在这里进行定义。
2. 软件开发。必须开发出符合规格说明的软件。
3. 软件确认。软件必须通过确认来确保软件所做的是客户所想要的。
4. 软件演化。软件必须通过演化来满足不断变化的客户需要。

这些活动本身也是复杂的活动，还会包括一些子活动，例如需求确认、体系结构设计、单元测试等。软件过程还包括其他一些活动，例如软件配置管理、项目计划等支持软件生产活动的活动。

当我们描述和讨论所谓的软件过程时，我们总是在谈论过程中的活动（如数据模型的定义、用户界面设计等）以及这些活动的顺序。我们可以将过程与人们开发软件所做的所有事情联系到一起。然而，当描述过程时，重要的是描述涉及哪些人、产生了什么以及影响活动序列的条件，具体如下。

1. 产品交付物是软件过程活动的产出物。例如，体系结构设计活动的产出物是软件体系结构模型。

2. 角色反映了参与过程的人在其中的职责。角色的例子包括项目经理、配置经理、程序员等。

3. 前置和后置条件是指在一个过程活动执行之前和之后或者产品生产之前或之后必须满足的条件。例如在体系结构设计开始之前，一个前置条件可能是客户已经认可了所有的需求；在此活动结束之后，一个后置条件可能是描述体系结构的UML模型已经进行了评审。

软件过程是复杂的，而且像所有智力和创造性过程一样，依赖于人们的主观决策和判断。由于并不存在放之四海而皆准、适用于所有类型软件的过程，大多数软件企业都制定了

自己的开发过程。软件过程在不断演化以充分利用组织中的软件开发者的能力以及所开发的系统的特性。对于安全攸关系统，需要一个非常结构化的开发过程，其中要保存详细的记录。对于业务系统，由于需要适应快速变化的需求，因此一个更加灵活、敏捷的过程可能更好。

如第1章所述，专业软件开发是一个受管理的活动，因此计划是所有过程的一个固有部分。计划驱动的过程是提前计划好所有的过程活动，然后按计划去衡量进度。在敏捷过程（见第3章）中，计划是在软件开发过程中增量和持续进行的，这样就可以很容易调整过程以反映不断变化的客户或产品需求。正如 Boehm 和 Turner（Boehm and Turner 2004）所说，每种方法适合于不同类型的软件。通常，对于大系统，你需要在计划驱动的过程和敏捷过程之间做出权衡。

虽然没有放之四海而皆准的软件过程，许多组织中仍然存在很大的软件过程改进空间。这些组织的过程中可能包括落后的技术，或者没有充分利用工业界的许多软件工程最佳实践。甚至许多组织还没有在自己的软件开发过程中系统性地采用软件工程方法。他们可以通过引入 UML 建模和测试驱动的开发等技术来改进自己的过程。本章后面将简要介绍软件过程改进，而在线章节中的第26章中进一步详细介绍了软件过程改进。

2.1 软件过程模型

如第1章所述，软件过程模型（有时也称为软件开发生命周期或 SDLC 模型）是软件过程的简化表示。每个过程模型都是从一个特定的侧面表现软件过程，所以只提供过程的部分信息。例如，过程活动模型描述了活动和它们的顺序，但是可能表现不出人们在这些活动中的角色。这一节中将介绍几个非常通用的过程模型（有时也叫作过程范型（process paradigms）），并从一个体系结构的视角呈现这些过程模型，即只关心过程的框架而忽略过程活动的细节。

这些通用过程模型是软件过程的高层和抽象描述，能用于解释不同的软件开发方法。你可以将它们视为一种过程框架，可以通过扩展和调整来创建更加特定的软件工程过程。

本章讨论的几个通用过程模型如下。

1. 瀑布模型。该模型包含了基本的过程活动（即规格说明、开发、确认、演化），并将它们表示为独立的过程阶段，例如需求规格说明、软件设计、实现、测试。

2. 增量式开发。该方法使得规格说明、开发和确认活动交错进行。系统开发体现为一系列的版本（增量），每个版本在前一版本的基础上增加一些功能。

3. 集成和配置。该方法依赖于可复用的构件或系统。系统开发过程关注在新的使用环境中配置这些构件并将它们集成为一个系统。

如前所述，没有放之四海而皆准、适用于所有不同类型软件开发的过程模型。正确的过程取决于客户和管理需求、软件使用所处的环境，以及所开发的软件类型。例如，一个安全攸关软件通常使用瀑布过程进行开发，因为在实现开始前需要进行大量的分析和文档化。软件产品现在总是用增量的过程模型来开发。业务系统正越来越多地通过配置和集成已有的系统来构造具有所需功能的新系统。

实践中的软件过程大多数都建立在通用过程模型基础上，但经常会结合其他模型的特性。大规模系统工程尤其如此。对于大规模系统，将所有通用过程中的一些最好的特性结合在一起是有意义的。你必须知道系统的核心需求，设计系统的软件体系结构以支持这些需

求。这是不能增量式开发的。这些大系统中的子系统可以使用不同的开发方法。系统中的一些部分已经得到了很好的理解，可以使用基于瀑布模型的过程进行规格说明和开发，或者可以作为成品系统买入并进行配置。而那些很难提前清楚刻画的部分则要采用增量的开发方法。这两种情况下，软件构件都很有可能得到复用。

有一些组织和人试图在所有这些通用模型基础上开发出一种“普遍适用”的过程模型。这些“普遍适用”的过程模型中最广为人知的一个是由美国的软件工程公司 Rational 开发的 Rational 统一过程 (Rational Unified Process, RUP) (Krutchen 2003)。RUP 是一种灵活的模型，可以通过不同的方式进行实例化以创建与这里所介绍的任何一个通用过程模型相似的过程。RUP 在一些大型软件公司 (特别是 IBM) 得到了采用，但并没有被广泛接受。



Rational 统一过程

Rational 统一过程 (Rational Unified Process, RUP) 将这里介绍的所有通用过程模型的元素聚集在一起，支持原型构造和软件的增量交付 (Krutchen 2003)。RUP 通常可以从 3 个视角进行描述：动态视角显示模型在时间上的各个阶段；静态视角显示过程活动；实践视角推荐过程中可以使用的好的实践。RUP 的阶段包括：初识阶段，将建立系统的业务案例；细化阶段，将开发需求和体系结构；构造阶段，将对软件进行实现；转移阶段，将对软件进行部署。

<http://software-engineering-book.com/web/rup/>

2.1.1 瀑布模型

最早出现的软件开发过程模型起源于大型军事系统工程中所使用的工程过程模型 (Royce 1970)。该模型将软件开发过程表示为一些阶段，如图 2-1 所示。由于该模型从一个阶段流动到另一个阶段，因此该模型被广泛称为瀑布模型或者软件生命周期模型。瀑布模型是计划驱动的软件过程的一个例子。原则上，至少应该在软件开发开始之前对所有的过程活动进行计划和进度安排。

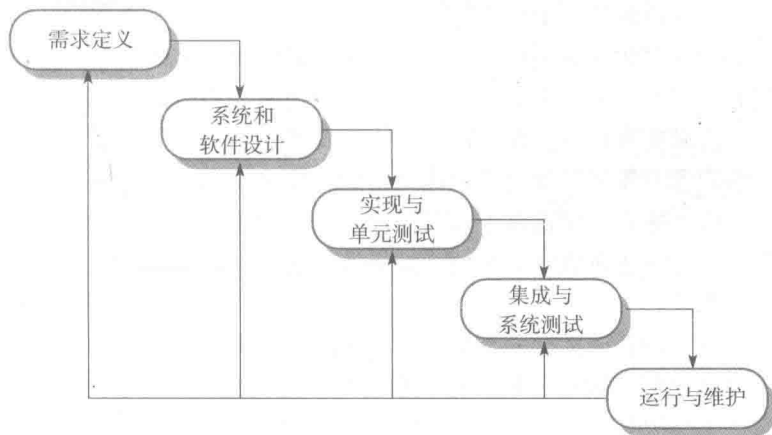


图 2-1 瀑布模型

瀑布模型中的阶段直接反映以下这些基本的软件开发活动。

1. 需求分析和定义。通过咨询系统用户建立系统的服务、约束和目标，详细定义这些信息并作为系统的规格说明。

2. 系统和软件设计。系统设计过程将需求分配到硬件或软件系统上，建立一个总体的系统体系结构。软件设计涉及识别并描述基本软件系统抽象以及它们之间的关系。

3. 实现和单元测试。在此阶段，将软件设计实现为一组程序或程序单元。单元测试验证每个单元是否满足其规格说明。

4. 集成和系统测试。各个程序单元或程序被集成为一个完整的系统，并进行测试以保证软件需求都得到了满足。测试之后，软件系统被交付给客户。

5. 运行和维护。通常这是时间最长的一个生命周期阶段。系统被安装并投入实践使用。维护包括修复没有在生命周期早期阶段发现的错误、改进系统单元的实现、随着新需求的发现而对系统的服务进行提升。



Boehm 的螺旋过程模型

Barry Boehm，软件工程的先驱之一，提出了一种风险驱动的增量过程模型。这个模型将过程表示为一个螺旋而不是活动序列（Boehm 1988）。

螺旋中的每一个循环表示软件过程中的一个阶段。这样，最里面的循环可能关注系统可行性，接下来的循环关注需求定义，然后是系统设计，等等。螺旋模型将变更避免与变更容忍结合到了一起。它假设变更是项目风险的结果，因此包含了显式的风险管理活动来降低这些风险。

<http://software-engineering-book.com/web/spiral-model/>

原则上，瀑布模型中每个阶段的结果是一个或多个审批通过（批准）的文档。后续的阶段在前一阶段结束前不应该开始。对于包含很高的制造成本的硬件开发这是有合理的。然而，对于软件开发，这些阶段存在重叠并相互反馈信息。在设计过程中可能发现需求的问题；在编码过程中可能发现设计的问题等。实践中的软件过程从来就不是一个简单的线性模型，而是包含从一个阶段到另一个阶段的反馈。

如果一个过程阶段中出现新的信息，那么此前阶段产生的文档应当进行修改以反映所需的系统变化。例如，如果发现一个需求实现起来代价很高，那么需求文档应当进行修改以移除该需求。然而，这需要客户同意并会延迟总体开发过程。

其结果是，客户和开发者可能过早地冻结了软件规格说明，从而使其不会发生进一步的变化。不幸的是，这意味着问题遗留下来待后面解决、问题被忽略或规避。过早地冻结需求可能意味着系统不会实现用户想要的东西。如果通过实现上的小技巧规避设计问题，那么可能导致结构很差的系统。

在最终的生命周期阶段（运行和维护）中，软件投入使用。在此阶段会发现最初的软件需求中的一些错误和遗漏。程序和设计错误会出现，还可能会产生新的功能性需求。因此，系统必须通过演化来保持有用。实施这些变更（软件维护）可能要重复进行此前的过程阶段。

在现实中，软件必须在开发过程中保持灵活并容纳变更。瀑布模型要求早期的承诺并且

在实施变更时要进行系统返工。这意味着瀑布模型只适用于以下这些类型的系统。

1. 嵌入式系统，其中软件必须与硬件系统连接和交互。由于硬件不灵活，将软件功能的决策推迟到开发时通常不可行。

2. 关键性系统，要求对软件规格说明和设计的安全性和信息安全进行全面的分析。在这些系统中，规格说明和设计文档必须完整，使得这些分析成为可能。在实现阶段修复规格说明和设计中与安全相关的问题通常非常昂贵。

3. 大型软件系统，属于更广阔的由多家合作企业共同开发的工程化系统的一部分。系统中的硬件可能使用相似的模型开发，相关企业发现使用同样的模型进行硬件和软件开发更容易一些。而且，由于涉及多家企业，可能需要完整的规格说明以使不同的子系统可以独立开发。

如果团队沟通可以采取非正式的方式并且软件需求快速变化，那么瀑布模型并不是正确的过程模型选择。迭代化的开发以及敏捷方法对于这些系统是更好的选择。

瀑布模型的一个重要的变体是形式化的系统开发，其中会创建系统规格说明的数学模型。该模型接下来会通过使用保持一致性的数学变换精化为可执行的代码。形式化的开发过程，例如基于B方法（Abrial 2005, 2010）的过程，主要用于有着严格的安全性、可靠性或信息安全需求的软件系统开发。形式化方法简化了安全性或信息安全案例的产生。这向客户或监管者显示了系统实际上满足它的安全性或信息安全需求。然而，由于开发形式化规格说明很昂贵，这一开发模型除了关键性系统工程外很少被使用。

2.1.2 增量式开发

增量式开发的思想是先开发出一个初始的实现，然后从用户那里获取反馈并经过多个版本的演化直至得到所需要的系统（见图2-2）。规格说明、开发和确认等活动不是分离的而是交织在一起，活动之间存在快速的反馈。在某种形式上，增量开发现在已经成为最常用的应用系统和软件产品开发方法。该方法可以是计划驱动的、敏捷的，或者更为常见的是这些方法的混合。在计划驱动的增量式开发中，系统的增量是提前确定的；如果采用敏捷方法，那么早期的增量是确定好的，但后面的增量开发则取决于进度和客户优先级。

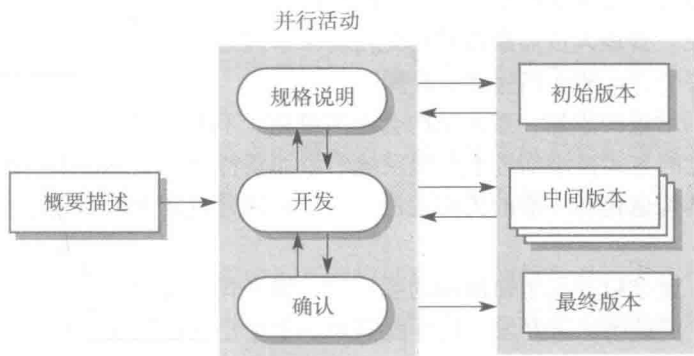


图 2-2 增量式开发

对于开发过程中需求很容易发生变化的系统而言，增量式软件开发（敏捷方法的一个基础构成部分）是一个比瀑布模型方法更好的选择。大部分业务系统和软件产品都是这样的。增量式开发反映了我们解决问题的方式。我们很少提前制定出完整的问题解决方案，而是逐

步地逼近解决方案，当我们意识到错误的时候则会进行回溯。通过增量式地开发软件，在开发过程中进行变更的成本会更低也更容易。

系统的每一个增量或版本都包括用户需要的一部分功能。通常，系统的早期增量包括最重要或最紧急的功能性需求。这就意味着客户或用户可以在相对比较早的阶段对系统进行评估，以确定系统是否提供了所需要的需求。假如不满足需要，那么只需要对当前的增量进行变更，也有可能为后续的增量定义新功能。

增量式开发与瀑布模型相比有以下 3 个主要的优势。

1. 降低了实现需求变更的成本。较之瀑布模型，重新分析和修改文档的工作量要少很多。
2. 在开发过程中更容易得到客户对于已完成的开发工作的反馈意见。客户可以对软件的演示版本进行评价，并可以看到需求已经实现了多少。客户通常都会感觉从软件设计文档中判断项目进度很困难。
3. 即使并未将所有的功能包含其中，也使得在早期向客户交付和部署有用的软件成为可能。与瀑布模型相比，客户可以更早地使用软件并从中获得价值。



增量式开发的问题

尽管增量式开发有很多优点，但也存在一些问题。导致困难的主要原因是，大型组织的官僚办事规程随着时间的推移已经根深蒂固，这些办事规程与更加灵活的迭代和敏捷过程可能不太不匹配。

有时候这些办事规程是有其存在的充分理由的，例如有些规程可能是为了确保软件开发过程能严格遵守外部法规（例如美国的萨班斯-奥克斯利会计准则）。这些规程可能是无法改变的，因此过程冲突是不可避免的。

<http://software-engineering-book.com/web/incremental-development/>

从管理的角度看，增量式开发方法存在两个问题：

1. 过程不可见。管理人员需要常规的交付物来掌握进度。如果系统是快速开发的，那么要产生反映系统每个版本的文档就很不合算。
2. 伴随着新的增量的添加，系统结构会逐渐退化。不断的修改导致凌乱的代码，因为新需求会以任何可能的方式被添加进来。向系统中添加新特性将变得越来越困难，成本也越来越高。为了缓解结构退化和一般的代码混乱，敏捷方法建议定期对软件进行重构（改进和结构调整）。

在面对大型、复杂以及长生命周期的系统时，增量式开发的问题变得更加突出。这类系统的不同部分由不同的团队来开发，这样就需要一个稳定的框架或体系结构，负责系统不同部分的各个团队的职责需要按照体系结构来明确、清晰地定义。这种体系结构要求事先进行计划而不是增量地开发。

增量式开发并不意味着开发者必须向系统客户交付每一个增量。开发者可以增量地开发系统并向客户和其他相关利益相关者进行展示以获得评论和反馈，不一定需要将其交付给客户并在客户环境中进行部署。增量的交付（在 2.3.2 节中介绍）意味着软件在真实的运行过程

中被使用,因此用户反馈可能会更真实。然而,提供这种反馈并不总是可行的,因为试验新软件可能会干扰常规的业务过程。

2.1.3 集成与配置

大多数软件项目中都存在一定程度的软件复用。复用经常在参与项目开发的人了解到或查找与需要开发的功能相似的代码时以一种非正式的方式发生。他们寻找可复用的代码,按照需要对它们进行修改,并将它们与已开发的新代码相集成。

这种非正式复用的发生与所使用的开发过程无关。然而,自2000年开始,关注复用已有软件的软件开发过程已经得到了广泛的使用。面向复用的方法依赖于一个可复用的软件构件库以及一个用于构件组装的集成框架。

以下3类软件构件经常被复用。

1. 经过配置后可以在特定环境中使用的独立应用系统。这些系统是拥有很多特性的通用系统,但它们必须在特定的应用中进行调整和适配。
2. 作为一个构件或一个包开发的并且将与一个构件框架(例如Java Spring框架(Wheeler and White 2013))相集成的一组对象。
3. 按照服务标准并且可以通过互联网进行远程调用的Web服务。

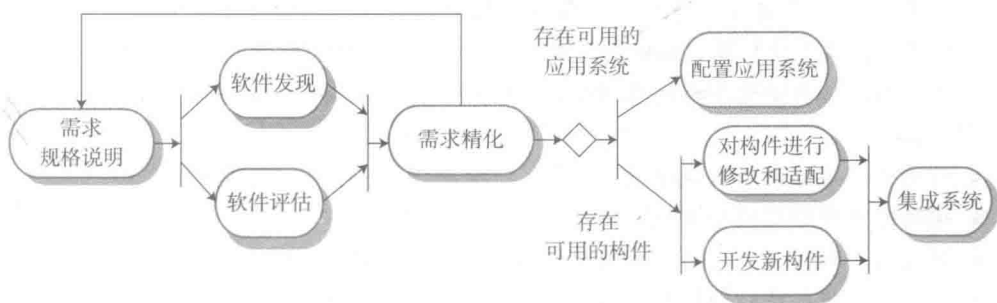


图 2-3 面向复用的软件工程

图2-3描述了一个通用的基于复用以及集成和配置的开发过程模型。该过程中包括以下这些阶段。

1. 需求规格说明。系统的初始需求被提出。这些需求不需要进行细化,但是应当包含对于基本的需求以及想要的系统特性的简要描述。

2. 软件发现和评估。给定软件需求的概要后,搜索提供所需要的功能的构件和系统。然后对候选的构件和系统进行评估,确定它们是否满足相应的基本需求以及是否适合用于当前系统。

3. 需求精化。此阶段利用所发现的可复用构件和应用的信息对需求进行精化。对需求进行修改以反映可用的构件,而系统规格说明会被重新定义。如果无法进行修改,那么可能会重新进入构件分析活动来查找替代的解决方案。

4. 应用系统配置。如果一个满足需求的成品应用系统是可用的,那么可以对该系统进行配置后使用以创建新系统。

5. 构件适配和集成。如果没有成品系统,那么可以对各个可复用构件进行修改并开发新构件。接着,这些构件被集成到一起来创建系统。

基于配置和集成的面向复用的软件工程在降低软件开发量以及降低成本和风险方面有着明显的优势。该方法通常还可以实现更快的软件交付。然而,需求权衡是不可避免的,这可能导致系统不完全满足用户的真实需求。此外,采用面向复用的开发方法还会失去一些对系统演化的控制,因为可复用构件的新版本并不在使用该构件的组织控制之下。

软件复用很重要,因此本书第三部分的一些章节还会对这一话题进行介绍。其中,第15章介绍了一些关于软件复用的一般性问题,第16和17章介绍了基于构件的软件工程,第18章介绍面向服务的系统。

2.2 过程活动

真实的软件过程中技术活动、协作活动以及管理活动交织在一起,其总体目标是完成一个软件系统的规格说明、设计、实现和测试。一般而言,现在的软件过程都是工具支持的。这意味着软件开发者可以使用很多软件工具来帮助自己的工作,例如需求管理系统、设计建模编辑器、程序编辑器、自动测试工具、调试器等。



软件开发工具

软件开发工具是用来支持软件工程过程活动的程序。这些工具包括需求管理工具、设计编辑器、重构支持工具、编译器、调试器、缺陷追踪工具、系统构建工具等。

软件工具通过自动化某些过程活动以及提供所开发软件的信息来提供软件过程支持。

例如:

- 作为需求规格说明或软件设计中的一部分的图形化系统模型的开发;
- 通过这些图形化模型生成代码;
- 通过用户利用交互式的方式所创建的图形化界面描述来生成用户界面;
- 通过提供正在执行的程序的信息进行程序调试;
- 将使用一种编程语言的某个旧版本所编写的程序翻译为一个较新的语言版本。

多种工具可以被整合在一个被称为交互式开发环境(Interactive Development Environment, IDE)的框架中。该环境提供了一组通用的基础设施,使得不同的工具可以使用这些基础设施更容易地进行通信并以一种集成的方式运行。

<http://software-engineering-book.com/web/software-tools/>

4个基本的过程活动——规格说明、开发、确认、演化,在不同的开发过程中的组织方式也各不相同。在瀑布模型中,这些活动被组织为一个序列;而在增量式开发中,这些活动交织在一起。这些活动如何开展取决于所开发的软件的类型、开发者的经验和能力,以及开发此软件的组织的类型。

2.2.1 软件规格说明

软件规格说明或需求工程过程的目的是理解和定义系统需要提供哪些服务,以及识别对于系统开发和运行的约束。需求工程是软件过程中一个特别关键的阶段,这个阶段所犯的错误将不可避免地在后续的系统设计和实现阶段造成问题。

需求工程过程开始之前,企业可能会进行一个可行性或市场研究,以评估是否存在与该软件相应的需要或者市场,以及开发所要求的软件是否在技术上和财务上可行。可行性研究是短期的、相对低成本的研究,为是否进一步进行更详细的分析决策提供依据。

需求工程过程(图 2-4)的目的是产生一个得到共识的需求文档,其中刻画了一个满足利益相关者需求的系统。需求通常在两个细节层面上进行陈述。最终用户和客户需要一个需求的高层陈述;系统开发者需要一个更详细的系统规格说明。

需求工程活动中有以下 3 个主要活动。

1. 需求抽取与分析。该过程通过观察已有的系统、与潜在的用户和采购方进行讨论、任务分析等手段,得出系统需求。此过程中可能会开发一个或多个系统模型和原型。这些都会有助于理解所刻画

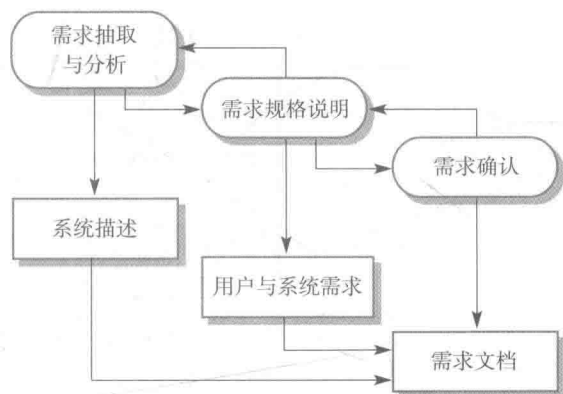


图 2-4 需求工程过程

2. 需求规格说明。需求规格说明活动将需求分析中所收集的信息转化为定义一组需求的文档。该文档中可能包含两种类型的需求:用户需求,是面向系统客户和最终用户的系统需求的抽象陈述;系统需求,是对将提供的功能的一种更加详细的描述。

3. 需求确认。该活动检查需求的现实性、一致性和完整性。在此过程中会不可避免地发现一些需求文档中的错误。必须对需求文档进行修改以纠正这些问题。

需求分析在定义和规格说明过程中持续进行,并且在整个过程中都有可能出现新需求。因此,分析、定义和规格说明活动是相互交织的。

在敏捷方法中,需求规格说明不是一个独立的活动,而被视为系统开发的一部分。针对每个系统增量的需求,在该增量开发开始之前才会进行非正式的刻画。需求根据用户的优先级进行刻画。需求的抽取来自于用户,他们是开发团队的一部分或者与开发团队一起紧密工作。

2.2.2 软件设计和实现

软件开发的实现阶段是开发一个可执行的系统以交付给客户的过程。有时候软件设计和编码活动会分开。然而,如果使用敏捷方法,那么设计和实现是交织在一起的,不会在此过程中产生正式的设计文档。当然,仍然需要对软件进行设计,但是设计是以一种非正式的方式记录在白板和程序员的笔记本上的。

软件设计是对将要实现的软件的结构、系统所使用的数据模型和结构、系统构件间的接口的描述,有时候还会包括所用的算法。设计者不是立即完成全部设计的,而是分阶段完成设计的。设计者在设计过程中要不断添加所需的细节,同时不断修改先前的设计方案。

图 2-5 是设计过程的抽象模型,显示了设计过程的输入、过程活动以及过程的输出。设计过程活动既存在交织又相互依赖。关于设计的新信息不断生成,这会影响此前的设计决策。因此,设计的修改是不可避免的。

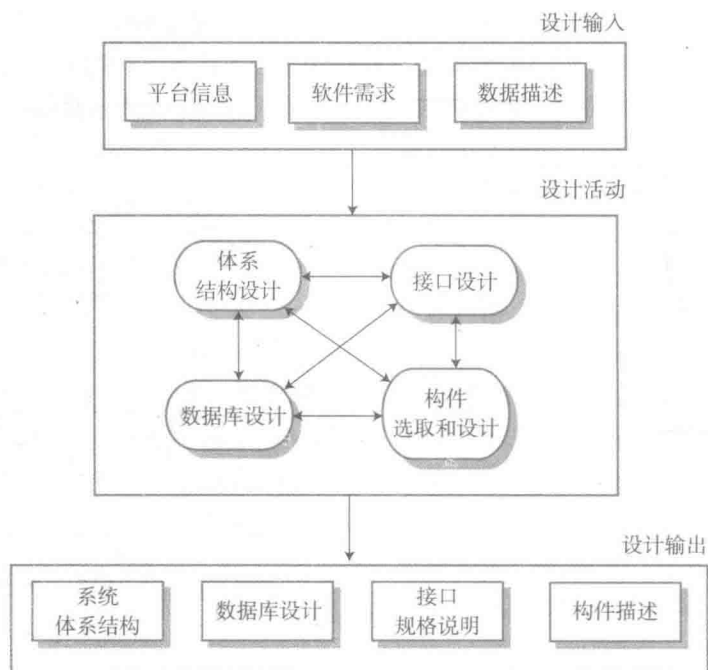


图 2-5 设计过程的通用模型

大多数软件都会与其他软件系统交互。这些其他系统包括操作系统、数据库、中间件和其他应用系统。这些构成了所谓的“软件平台”，即软件将会在其中运行的环境。关于该平台的信息对于设计过程是一种重要的输入，因为设计者必须决定如何以最好的方式将系统与其环境集成在一起。如果系统要对已有的数据进行处理，那么对于这些数据的描述可以包含在平台规格说明中。否则，数据描述必须作为输入提供给设计过程，使得设计者可以定义系统数据的组织。

不同开发项目中的设计过程活动有所相同，取决于所开发的系统类型。例如，实时系统要求一个额外的时间设计阶段，但可能不包含数据库，因此可以没有数据库设计阶段。图 2-5 中显示了信息系统设计过程中可能包含的 4 个活动。

1. 体系结构设计。将识别系统的总体结构、基本的构件（有时候也称为子系统或模块）、它们之间的关系以及它们是如何分布的。

2. 数据库设计。将设计系统的数据结构以及数据库中的数据表示方式。同样，这里的工作取决于是否要复用已经存在的数据库或者创建一个新的数据库。

3. 接口设计。将定义系统构件间的接口。接口规格说明必须是无歧义的。有了精确的接口定义，一个构件就可以在无须了解另一个构件的具体实现的情况下使用该构件。针对接口规格说明达成一致后，各个构件就可以独立进行设计和开发了。

4. 构件选取和设计。将搜索可复用的构件，如果没找到合适的构件那么就设计新的软件构件。此阶段的设计可能只是对构件进行简单的描述，而把实现细节留给程序员。也可以是确定要对一个可复用构件进行的一个修改列表，或者是用 UML 表示的一个详细的设计模型。接下来，设计模型可以被用于自动地生成实现。

这些活动产生设计输出，这在图 2-5 中也有表示。对于关键性系统，设计过程的输出是详细设计文档，其中设定了精确和准确的系统描述。如果使用模型驱动的开发方法（见第 5

章),设计输出是设计图。如果使用敏捷开发方法,设计过程的输出可能不是独立的规格说明文档,而是在程序的代码中进行表示。

开发一个程序以实现一个系统很自然地会遵循系统设计。虽然有些类型的程序(例如安全攸关系统)通常在实现开始前会进行详细的设计,更常见的情况是设计和程序开发交织在一起。可以使用软件开发工具来从一个设计生成一个骨架程序,其中包括定义和实现接口的代码,很多时候开发人员只需要添加每个程序构件的操作的实现细节。

编程是一个个人化的活动,没有可以广泛遵循的一般性的过程。有些程序员从他们理解得比较好的构件开始,开发完这些再转而对理解得没那么好的构件进行开发。其他一些程序员则相反,将熟悉的构件留到最后,因为他们知道如何开发这些构件。一些开发者喜欢在开发中早早地定义数据,然后使用数据来驱动程序开发;其他开发者则将数据定义尽可能延后。

通常,程序员会对已经开发好的代码进行一些测试。这经常可以发现一些必须从程序中移除的程序缺陷(或 bug)。找到并修复程序缺陷被称为调试(debug)。缺陷测试和调试是不同的过程:测试确定缺陷的存在;调试关注定位并修正这些缺陷。

在进行调试时,你必须针对可观察的程序行为建立一些假设,然后测试这些假设以找到导致输出异常的缺陷根源。测试假设可能要手动跟踪程序代码,还可能会需要新的测试用例来定位问题。交互式的调试工具可以显示程序变量的中间值以及所执行的语句轨迹,通常可以用于支持调试过程。

2.2.3 软件确认

软件确认,或更一般性地,验证和确认(Verification and Validation, V&V),目的是确定系统是否符合它的规格说明,同时是否符合系统客户的期望。程序测试,即用模拟的测试数据运行系统,是最基本的确认技术。确认还可以在从用户需求定义到程序开发的每一个软件过程阶段中包含检查性的过程(例如审查和评审)。然而,大部分验证和确认的时间和工作都是花在程序测试上。

除了一些小程序,系统不应当作为一个巨大的单元整体进行测试。图 2-6 描述了一个三阶段的测试过程,其中系统构件各自进行测试然后对集成后的系统进行测试。对于定制化软件,客户测试中会使用真实客户数据对系统进行测试。对于作为应用销售的产品,客户测试有时被称为 β 测试,即经过挑选的用户对软件进行试用和评价。



图 2-6 测试阶段

测试过程包括以下这些阶段。

1. 构件测试。由系统的开发人员对组成系统的构件进行测试。每个构件都在其他构件不参与的情况下单独进行测试。构件可能是简单的实体,例如函数或对象类,也可能是这些实体构成的内聚的分组。构件测试通常都会使用自动化测试工具(例如面向 Java 的 JUnit),这些工具能够在构件的新版本被创建的时候重新运行测试(Koskela 2013)。

2. 系统测试。系统构件被集成到一起创建一个完整的系统。这一过程主要关注找到由于非预期的构件间交互和构件接口问题所导致的错误。该过程也关注系统是否满足相应的功能性和非功能性的需求,并测试系统的涌现特性。对于大型系统而言,这可能是一个多阶段的

过程,在此过程中构件首先被集成以形成子系统并分别进行测试,然后这些子系统再被集成以形成最终的系统。

3. 客户测试。这是系统被接受并投入运行之前的测试过程中的最后阶段。系统由系统的客户(或潜在的客户)而不是模拟的测试数据来进行测试。对于定制化构造的软件,客户测试可能会发现系统需求定义中的错误和遗漏,因为基于真实数据的系统运行与基于测试数据的系统运行方式不一样。客户测试还可以发现其他一些需求问题,包括系统的设施无法真正满足用户需要或者系统的性能不可接受。对于产品,客户测试可以显示软件产品在多大程度上满足客户的需要。

理想情况下,构件缺陷可以在测试过程的早期被发现,而接口问题可以在系统集成时被发现。然而,当发现缺陷时,必须对程序进行调试,测试过程中的其他一些阶段可能需要重复进行。程序构件中的错误可能会在系统测试过程中暴露出来。因此,这是一个迭代化的过程,后面阶段中的信息会反馈给该过程早期的一些阶段。

通常,构件测试是常规开发过程的一部分。程序员构造自己的测试数据,并在代码开发过程中增量地测试代码。程序员了解所开发的构件,因此是生成测试用例的最佳人选。

如果采用增量式开发方法,每一个增量在开发的时候都要基于针对该增量的需求进行测试。在测试驱动的开发(敏捷过程的一个常规部分)中,测试在开发开始前与需求一起被开发出来。这可以帮助测试人员和开发人员理解需求,并确保测试用例的创建没有延迟。

当使用计划驱动的软件过程时(例如对于关键性系统开发),测试是由一组测试计划驱动的。一个独立的测试团队在基于系统规格说明和设计开发的测试计划基础上开展工作。图 2-7 描述了测试计划如何将测试和开发活动链接到一起。这个有时被称为开发的“V 模型”(把图转一下就可以看到 V)。“V 模型”显示了对应于瀑布过程模型中每个阶段的软件确认活动。

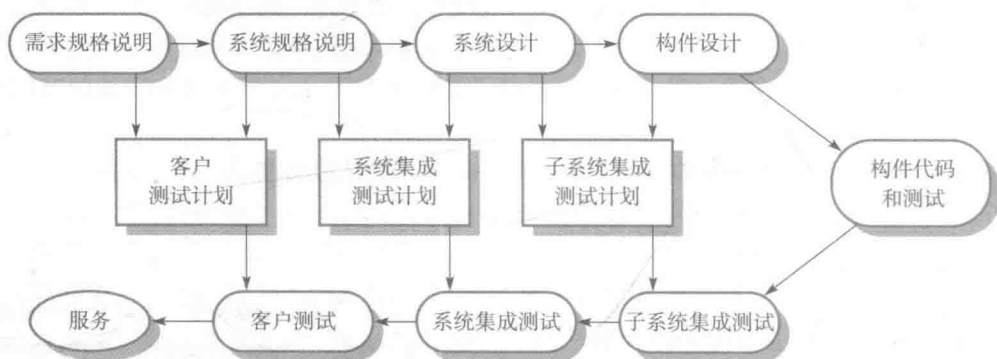


图 2-7 计划驱动的软件过程中的测试阶段

当一个系统要作为一个软件产品在市场上销售时,被称为 β 测试的测试过程经常会被用到。 β 测试向一些同意使用目标系统的潜在客户交付该系统。他们向系统开发者报告问题。这一过程将产品暴露在真实使用面前,并发现没有被产品开发者预见到的错误。在该反馈后,软件产品可以被修改并发布以进行进一步的 β 测试或一般性的销售。

2.2.4 软件演化

软件的灵活性是大型、复杂系统中包含越来越多的软件的主要原因之一。对于硬件而言,一旦做出了一个关于其制造的决定,再对硬件设计进行变更将是非常昂贵的。然而,软

件则可以在系统开发之中或之后的任何时间进行修改。即使非常大范围的修改也比对于系统硬件的相应变更便宜很多。

从历史上看,软件开发过程和软件演化(软件维护)过程之间总是分离的。人们认为软件开发是一个创造性的活动,其中软件系统从一个初始的概念开始进行开发直至得到一个可以工作的系统。然而,人们有时候会觉得软件维护比较平缓而且没有那么有趣。他们认为软件维护没有原始的软件开发那么有趣以及富于挑战性。

这种开发和维护之间的区分越来越不合时宜。现在很少有软件系统是全新的系统,将开发和维护视为一个连续的过程无疑是更合理的。与将开发和维护视为两个分离的过程的认知相比,一种更加现实的认知是将软件工程视为一个演化式的过程(图2-8),其中软件在其生命周期中随着不断变化的需求和客户需要而持续发生变化。

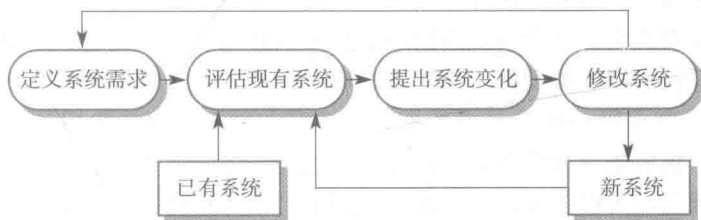


图 2-8 软件系统演化

2.3 应对变化

对于所有的大型软件项目而言变化都是无法避免的。系统需求随着业务应对外部压力、竞争和管理优先级的需要等变化而发生变化。当可用的新技术出现时,新的设计和实现方法成为可能。因此,不管使用什么样的软件过程模型,支持对所开发的软件进行变更都是基本要求。

变化增加了软件开发成本,因为变化通常意味着已经完成的工作必须重做。这被称为返工(rework)。例如,如果对一个系统的需求之间的关系进行了分析并识别出了新的需求,那么需求分析中的一些或全部工作必须重复执行。为此,可能有必要重新设计系统以支持新需求、修改已经开发好的一些程序,并且重新对系统进行测试。

以下这两个相关的方法可以用来降低返工的成本。

1. 变化预测。软件过程包括可以在要求大量返工之前预见或预测可能的变化的活动。例如,可以开发一个原型系统来向客户显示系统的一些重要特性。客户可以使用原型进行试验,并在花费高额的软件生产成本之前对需求进行精化。
2. 变化容忍。通过过程和软件设计使得对系统的修改很容易进行。这通常包括某种形式的增量开发。所提出的变更可能是在还没有开发的增量上实现。假如无法做到这一点,那么只需要修改一个增量(系统的一小部分)来实现变化。

这一节将介绍如下两种应对变化以及修改系统需求的方法。

1. 系统原型。系统或系统的一部分的一个版本被快速开发以检验客户需求以及设计决策的可行性。这是一种变化预测方法,因为它允许用户在交付前使用系统进行试验并据此精化他们的需求。因此,交付之后的需求变更请求的数量很有可能会减少。

2. 增量交付。系统的增量被交付给客户进行评论和试验。这种方法既支持变化避免又可

以支持变化容忍。避免了交付不成熟的需求实现，并允许在之后的增量中进行改变且将成本控制在较低水平。它避免了对于系统整体需求的不成熟的承诺，并且允许变化能够以较低的代价加入到后续的增量中。

重构的概念，即改进程序的结构和组织，也是一种重要的支持变化容忍的机制。本书将在第3章（敏捷方法）中介绍重构。

2.3.1 原型

原型是一种软件系统的早期版本，用于演示概念、尝试候选设计方案、更好地理解问题以及可能的解决方案。快速、迭代化的原型开发十分重要，这样就可以控制成本，而系统的相关利益相关者也可以在软件过程的早期试用系统原型。

软件原型可以用于软件开发过程以帮助对可能需要的变化进行预测：

1. 在需求工程过程中，原型可以帮助对系统需求进行抽取和确认。
2. 在系统设计过程中，原型可以用于探索软件解决方案，并用于系统用户界面的开发。

系统原型使得潜在用户可以看到系统能够在多大程度上支持他们的工作。他们可以从中获得对于需求的新想法，并发现软件的长处和短处。然后他们可以提出新的系统需求。此外，随着原型的开发，人们还可以发现系统需求中的错误和遗漏。规格说明中所描述的某个特征可能看上去是清楚和有用的。然而，当该特征与其他功能相结合之后，用户经常会发现他们最初的观点是不正确或不完整的。这样就可以对系统规格说明进行修改以反映对需求的新理解。

可以利用系统原型在系统设计中设计试验，以便检查所提出的设计的可行性。例如，可以对一个数据库设计进行原型开发和测试，以检查它对于最常见的用户查询是否支持高效的数据访问。原型也是用户接口设计过程必不可少的一部分。最终用户参与的快速原型对于用户界面开发是唯一合理的方法。由于用户界面的动态特性，文本式的描述和图形无法充分表达用户界面需求和设计。

一个原型开发的过程模型如图2-9所示。原型开发的目的是在开发过程的一开始就应当明确。可能的目的包括开发用户界面、开发系统以确认系统功能性需求，或者开发一个系统来向管理人员展示应用。同一个原型无法满足所有的目的。如果不明确原型开发的目的，那么管理人员或最终用户可能会误解原型的功能。其结果是，他们可能无法获得本应从原型开发中得到的益处。

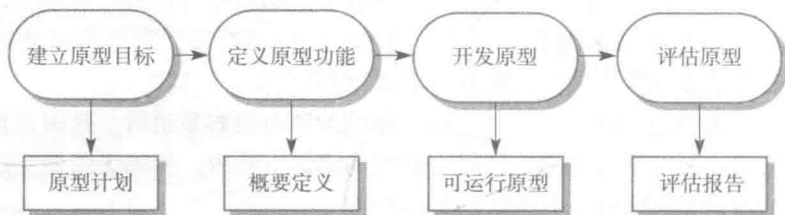


图 2-9 原型开发

过程的下一阶段就是决定哪些要放入原型系统中，或许更重要的是，哪些不放入原型系统中。为了减少原型开发的成本并加快交付进度，你可能要在原型中放弃一些功能。你可以决定放宽一些非功能性需求，例如响应时间、存储利用率等。错误处理和错误管理可以被略去，除非原型的目标是构建用户界面。程序的可靠性和质量方面的标准也可以降低。

该过程的最后一个阶段是原型评估。这个阶段必须安排用户培训，而且应该根据原型的目标制订一个评估计划。潜在的用户需要一段时间来适应新系统并进入正常使用模式。一旦他们正常使用该系统，他们就可以发现错误的以及被遗漏的需求了。原型一般都存在一个问题：用户可能不会像使用最终系统那样去使用原型。原型的测试者可能不是系统的典型用户。在原型评估过程中可能没有足够的时间。如果原型很慢，那么评估者可能会调整自己的工作方式并避免使用那些响应时间比较慢的系统特征。如果最终系统的响应时间能够大大改善，那么他们可能会以不同的方式使用系统。

2.3.2 增量式交付

增量式交付（如图 2-10 所示）是软件开发的一种方法，其中一部分被开发的增量会交付给客户并在他们的工作环境中进行部署和使用。在增量式交付过程中，客户定义哪些服务对他们最重要，哪些最不重要。在此基础上定义一系列交付增量，每个增量提供系统功能的一个子集。如何将服务分配到各个增量中取决于服务的优先级，其中优先级最高的服务首先被实现和交付。

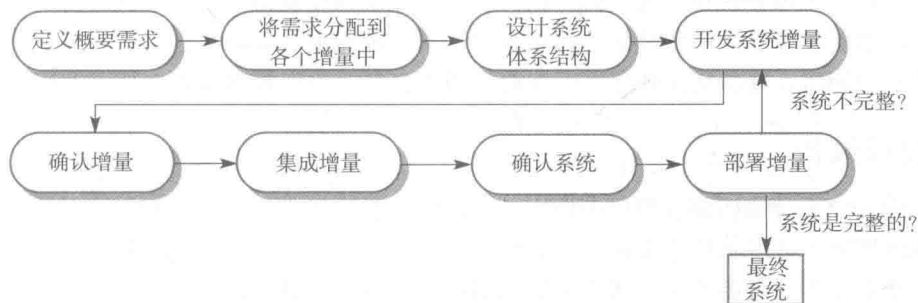


图 2-10 增量式交付

一旦确定了系统增量，将要在第一个增量中交付的服务的需求会被详细定义，并对第一个增量进行开发。在开发过程中，可以对后续增量进行进一步的需求分析，但是不接受对于当前增量的需求变更。

一旦完成并交付一个增量，就可以将其部署在客户的常规工作环境中。客户可以对系统进行试验，这可以帮助他们澄清他们对后续系统增量的需求。当新的增量完成后，再将它们与已有的增量相集成，使得每个交付的增量都能改进系统的功能。

增量式交付有下面这些优势。

1. 客户可以将早期的增量作为原型使用，从中获得关于后续系统增量的需求的经验。与原型不同的是，这些增量都是真实系统的一部分，因此当系统全部完成时用户不需要重新学习。

2. 客户不用等到整个系统交付就能从系统中获得价值。第一个增量可以满足他们最关键的需求，因此他们可以马上使用所开发的软件。

3. 这一过程保持了增量式开发的优点，那就是变更可以相对容易地加入到系统中。

4. 由于优先权最高的服务最先交付，然后后面的增量再被集成进来，这就使得最重要的系统服务的测试最充分。这意味着客户不太可能在系统最重要的部分上碰到软件失效。

然而，增量式交付也存在一些问题。在实践中，只有在引入一个全新系统并且系统的评

估者有足够的时间对新系统进行试验的情况下,这种方法才能较好地发挥作用。该方法的关键问题包括以下这些。

1. 当新系统准备替换一个已有系统时,迭代化交付会有问题。用户需要旧系统的所有功能,通常不愿意用一个不完整的新系统进行试验。同时使用旧系统和新系统经常是不现实的,因为它们很有可能会使用不同的数据库和用户界面。

2. 大多数系统需要一组由系统的不同部分使用的基础设施。由于在一个增量实现之前,相关的需求并没有详细定义,确定所有的增量都需要的公共基础设施可能很难。

3. 迭代化过程的本质是规格说明与软件一起开发。然而,这与许多组织的采购方式相冲突,这些采购方式要求将完整的系统规格说明作为系统开发合约的一部分。在增量方法中,直到最终的增量需求确定之前,都没有完整的系统规格说明。这要求一种新形式的合约,而大客户(例如政府部门)可能会难以接受。

对某些类型的系统而言,增量式的开发和交付不是最好的方法。例如,非常大型的系统涉及分布在地方不同地方的开发团队,一些嵌入式系统的软件取决于硬件开发,而一些关键性系统则要求对所有的需求进行分析以检查可能影响系统安全性或信息安全的交互。

当然,这些大型系统也会遇到同样的不确定以及需求变化的问题。因此,为了应对这些问题并获得增量式开发的一些优势,可以开发一个系统原型,并将其作为一个系统需求和设计的试验平台使用。基于从这种原型中获得的经验,针对最终的需求可以达成共识。

2.4 过程改进

如今的工业界持续面临着提供更便宜、更好的软件的压力,同时,交付时间压力也越来越大。其结果是,许多软件企业都寻求通过软件过程改进来提升软件的质量、降低成本、加速他们的开发过程。过程改进意味着理解当前的过程,并对其进行改变以提高产品质量,并且/或者降低成本和开发时间。Web 上的第 26 章对过程度量 and 过程改进相关的一些通用问题进行了详细介绍。

可以使用以下两种很不一样的方法来实现过程改进和变更。

1. 过程成熟度方法,关注改进过程和项目管理,并将好的软件工程实践引入到组织中。过程成熟度等级反映了好的技术和管理实践在多大程度上在组织的软件开发过程中得到了应用。这种方法的主要目标是提高产品质量和过程的可预测性。

2. 敏捷方法,关注迭代化的开发以及降低软件过程中的额外开销。敏捷方法的主要特点是快速交付功能以及对客户需求变更的快速响应。这里的改进哲学认为:最好的过程是那些额外开销最低的过程,而敏捷方法可以做到这一点。

第 3 章中将详细介绍敏捷方法。

拥护这两种方法其中之一的人们一般都会怀疑另一种方法的优势。过程成熟度方法来源于计划驱动的开发,因为会引入一些与程序开发并不直接相关的活动,通常会增加“额外开销”。敏捷方法关注所开发的代码,并特别将形式化和文档化进行最小化。

过程成熟度方法所基于的过程改进的一般过程是一个循环的过程,如图 2-11 所示。该过程中的阶段如下。

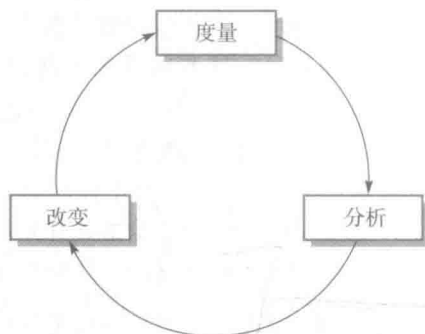


图 2-11 过程改进周期

1. 过程度量。对软件过程或产品的一个或多个属性进行度量。这些度量构成了一个基线，可以帮助你确定过程改进是否有效。当引入改进后，可以对同样的属性再次进行度量，它们很有可能已经有所改善。

2. 过程分析。对当前的过程进行评价，识别过程中的弱点和瓶颈。描述当前过程的过程模型（有时称为过程地图）可以在此阶段开发出来。分析可以关注考虑过程特性，例如过程的响应速度和鲁棒性。

3. 过程改变。提出对当前过程的改变方法以应对一些已识别出的过程弱点。引入变更后，改进循环继续收集与变化有效性相关的数据。

没有关于一个过程或者使用该过程所开发的软件的具体数据，就无法对过程改进的价值进行评价。然而，刚开始进行过程改进的企业很有可能没有过程数据作为改进基线。因此，作为第一个改变循环的一部分，可能必须收集关于软件过程的数据，并度量软件产品的特性。

过程改进是一个长期活动，因此改进过程中的每一个阶段都可能持续几个月。同时，过程改进也是一个持续的活动，因为任何新过程被引入后业务环境都会发生变化，新过程自身也必须随之进行演化以考虑这些环境变化。

过程成熟度的思想是在 20 世纪 80 年代后期被提出的。当时，软件工程研究所（Software Engineering Institute, SEI）提出了他们的过程能力成熟度模型（Humphrey 1988）。软件企业的过程成熟度反映了企业中的过程管理、度量以及好的软件工程实践的使用。这一思想被引入后，美国国防部就可以对国防项目承包商的软件工程能力进行评估，其目的是将软件开发合同给那些达到一定过程成熟度的承包商。该模型中包含 5 个过程成熟度等级，如图 2-12 所示。这些内容在过去 25 年中一直在演化和发展（Chrissis, Konrad, and Shrum 2011），但是 Humphrey 所提出的模型的基本思想仍然是软件过程成熟度评价的基础。

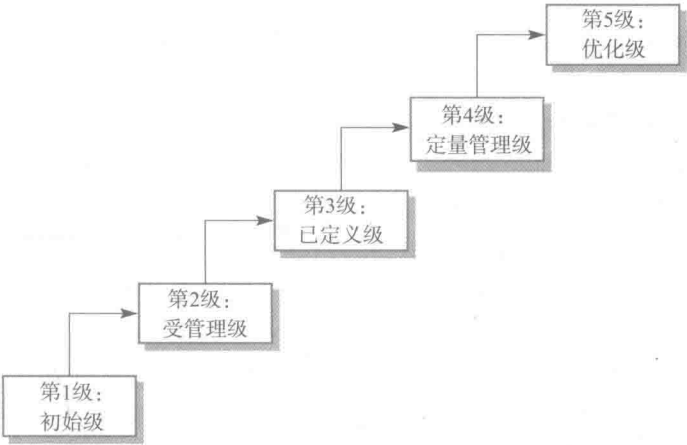


图 2-12 能力成熟度等级

过程成熟度模型包括以下几个等级。

1. 初始级。与过程域相关的目标令人满意。对于所有的过程，将要进行的工作范围得到了明确定义并与团队成员进行了沟通。

2. 受管理级。在这个等级上，与过程域相关的目标得到了满足，组织政策明确定义了每个过程应当在什么时候使用。必须有文档化的项目计划来定义项目目标。资源管理和过程监

控规程必须在整个机构中到位。

3. 已定义级。这个等级关注组织的标准化以及过程的部署。每个项目都有一个受管理的过程,该过程是在一组定义好的组织过程基础上按照项目需求进行适应性调整得到的。必须收集过程资产和过程度量,并用于未来的过程改进。

4. 量化管理级。在这个等级上,存在相应的组织职责,使用统计或其他定量方法来控制子过程。也就是说,所收集的过程和产品度量必须用于过程管理。

5. 优化级。在这个最高的等级上,组织必须使用过程和产品度量来驱动过程改进。必须对趋势进行分析,并根据不断变化的业务需要对过程进行调整。

在过程成熟度等级上的工作对软件产业有着巨大的影响。这些工作关注于已应用的并使软件工程能力得到显著改进的软件工程过程和实践。然而,对于小企业而言,正式的过程改进中的额外开销过高,而使用敏捷过程的成熟度评估也比较难。其结果是,只有大的软件企业现在还在软件过程改进中使用这一关注成熟度的方法。

要点

- 软件过程是生产一个软件系统过程中所包含的一系列活动。软件过程模型是这些过程的抽象表示。
- 通用过程模型描述软件过程的组织。这些通用过程模型的例子包括瀑布模型、增量式开发、可复用构件配置与集成等。
- 需求工程是开发软件规格说明的过程。规格说明的目的是向开发者传达客户方对于系统的需要。
- 设计和实现过程是将需求规格说明转换为一个可运行的软件系统的过程。
- 软件确认是检查系统是否符合它的规格说明以及是否符合系统用户的真实需要的过程。
- 软件演化发生在修改已有的软件系统以满足新的需求的时候。变化是持续的,软件必须演化以保持有用性。
- 过程应该包含应对变化的活动。可能包含一个原型构造阶段,这会有助于避免在需求和设计上的错误决策。过程可以按照迭代化开发和交付进行组织,这样变更可以在不影响系统整体的情况下进行。
- 过程改进是改进现有的软件过程以提高软件质量、降低开发成本、缩短开发时间的过程。过程改进是一个循环式的过程,包括过程度量、分析和改变。

阅读推荐

《Process Models in Software Engineering》是一本全面介绍各种软件工程过程模型的很棒的书。(W. Scacchi, Encyclopaedia of Software Engineering, ed. J. J. Marciniak, John Wiley & Sons, 2001) <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>

《Software Process Improvement: Results and Experience from the Field》一书汇集了一些关注过程改进案例研究的论文,其中的案例来自于挪威的一些中小企业。其中还包括对于过程改进的一些通用问题的很好介绍。(Conradi, R., Dybå, T., Sjøberg, D., and Ulsund, T. (eds.), Springer, 2006)

《Software Development Life Cycle Models and Methodologies.》这篇博客对几个已提出并使用的软件过程模型进行了简要总结。其中讨论了每个模型的优点和缺点。(M. Sami, 2012) <http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap2/>

支持视频的链接: <http://software-engineering-book.com/videos/software-engineering/>

练习

- 2.1 针对以下每个系统, 请推荐最合适的可以管理其开发的基础的通用软件过程模型, 按照所开发系统的类型给出你的理由。
 - 一个汽车中的防抱死刹车控制系统;
 - 一个支持软件维护的虚拟现实系统;
 - 一个准备替换现有系统的大学会计系统;
 - 一个交互式的旅行规划系统, 可以帮助用户以最小的环境影响规划旅程。
- 2.2 为什么增量式开发是开发业务软件系统的最有效的方法? 为什么这种模型不那么适用于实时系统工程?
- 2.3 考虑图 2-3 中所示的集成和配置过程模型。为什么在这个过程中要重复需求工程活动?
- 2.4 为什么在需求工程过程中区分用户需求开发和系统需求开发是重要的?
- 2.5 用一个例子解释为什么体系结构设计、数据库设计、接口设计、构件设计这些设计活动是相互依赖的。
- 2.6 为什么软件测试应当总是一种增量、分阶段的活动? 程序员是测试他们自己所开发的程序的最佳人选吗?
- 2.7 为什么在复杂系统中变化是不可避免的? 举出一些有助于预测可能的变化并使所开发的软件更适应变化的软件过程活动的例子 (除了原型和增量交付之外)。
- 2.8 假设你已经开发了一个软件系统原型, 且你的经理对其印象深刻。她提出应该将这个原型根据需要增加一些新特性后作为一个生产系统投入使用。这避免了系统开发的一些开销并能使系统可理解、可用。写一份简单的报告向你的经理解释为什么原型系统通常不应该作为生产系统使用。
- 2.9 指出 SEI 的能力成熟度框架中所包含的过程评估和改进方法的两个优点和两个缺点。
- 2.10 从历史上看, 新技术的出现经常导致劳动力市场的深远变革, 并使得人们的工作被取代 (至少是暂时性的)。讨论广泛的过程自动化的引入是否很有可能对软件工程师产生同样的后果。如果你不认为会如此, 解释为什么不会。如果你认为这样会减少工作机会, 那么受影响的工程师被动地或主动地抵制该技术的引入是否道德?

参考文献

Abrial, J. R. 2005. *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.

———. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, 21 (5), 61–72. doi:10.1145/12944.12948.

Boehm, B. W., and R. Turner. 2004. "Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods." In *26th Int. Conf on Software Engineering*, Edinburgh, Scotland. doi:10.1109/ICSE.2004.1317503.

Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed. Boston: Addison-Wesley.

Humphrey, W. S. 1988. "Characterizing the Software Process: A Maturity Framework." *IEEE Software* 5 (2): 73–79. doi:10.1109/2.59.

Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers*. Greenwich, CT: Manning Publications.

Krutchén, P. 2003. *The Rational Unified Process—An Introduction*, 3rd ed. Reading, MA: Addison-Wesley.

Royce, W. W. 1970. "Managing the Development of Large Software Systems: Concepts and Techniques." In *IEEE WESTCON*, 1–9. Los Angeles, CA.

Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.

敏捷软件开发

目标

本章的目标是介绍敏捷软件开发方法。阅读完本章后，你将：

- 理解敏捷软件开发方法的原理、敏捷宣言，以及敏捷和计划驱动的开发的区别；
- 了解重要的敏捷开发实践，例如用户故事、重构、结对编程、测试先行的开发；
- 理解面向敏捷项目管理的 Scrum 方法；
- 理解敏捷开发方法的伸缩性问题，以及在大型软件系统开发中将敏捷方法与计划驱动的方法相结合。

许多企业现在都在全球化、快速变化的环境中运行。它们必须响应新的机会和市场，响应不断变化的经济环境以及竞争产品和服务的出现。软件是几乎所有业务运行的一部分，因此新的软件必须快速开发出来以利用新的机会以及响应竞争压力。因此，快速的软件开发和交付对于大多数业务系统而言都是最关键的要求。事实上，如果可以快速部署一些重要的新软件，有时企业宁愿在软件质量和需求上进行权衡和妥协。

因为这些企业运行在一个不断变化的环境中，因此在实践中不可能定义出一个稳定的软件需求的完整集合。需求会发生变化，因为客户发现无法预测系统将会如何影响他们的工作实践、系统将如何与其他系统交互，以及哪些用户操作应该自动化。只有当一个系统已经被交付并且用户获得相关的系统经验之后，真实的需求才会变得清晰。即使到了这个时候，外部因素仍然会驱使需求变化。

计划驱动的软件开发过程希望完整地定义需求规格说明，然后对系统进行设计、构建和测试，这一思想并不适应于快速的软件开发。如果发现需求变化或者需求存在问题，那么系统设计和实现必须进行返工和重新测试。其结果是，传统的瀑布或基于规格说明的过程通常是一个冗长的过程，最终的软件要在最初定义规格说明后很长时间才能交付给客户。

对于某些类型的软件，例如安全攸关的控制系统，其中完整的系统分析必不可少，这种计划驱动的方法是正确的选择。然而，在一个快速变化的业务环境中，这一过程可能会导致真正的问题。当软件可以交付使用时，最初采购该系统的理由可能已经很快发生了变化，以至于软件实际上已经没什么用了。因此，特别是对于业务系统，关注快速软件开发和交付的开发过程是极其重要的。

对于快速软件开发以及可以处理需求变化的过程的需要多年前就已经被接受了（Larman and Basili 2003）。然而，随着“敏捷方法”的思想的发展，例如极限编程（Beck 1999）、Scrum 方法（Schwaber and Beedle 2001）和 DSDM 方法（Stapleton 2003），更快的软件开发实际上发生在 1990 年年底。

快速软件开发逐渐被称为敏捷开发或敏捷方法。这些敏捷方法的设计目的是快速产出有用的软件。所有已经提出的敏捷方法都具有如下这些共同的特性。

1. 规格说明、设计和实现过程交织在一起。没有详细的系统规格说明，设计文档化被最小化或者由用于实现系统的编程环境自动生成。用户需求文档是对最重要的系统特性的概览定义。

2. 系统按照一系列增量进行开发。最终用户和其他系统利益相关者参与每个增量的规格说明和评估。他们可能会提出对于软件的变更要求以及应当在系统的后续版本中实现的新需求。

3. 使用广泛的工具来支持开发过程。可以使用的工具包括自动化测试工具、支持配置管理和系统集成的工具、用户界面自动化构造工具。

敏捷方法是一种增量的开发方法，其中的增量都很小，并且通常每 2 ~ 3 周就创建新的系统发布版本并交付给客户。这些方法使客户参与开发过程，以便获得关于需求变化的快速反馈。他们通过使用非正式的交流来代替使用书面文档的正式会议，从而尽量减少文档化。

敏捷软件开发方法将设计和实现视为软件过程的中心活动，将其他开发活动（例如需求抽取和测试）融入设计和实现中。与之相对的是计划驱动的软件工程方法，该方法明确软件过程中各个独立的阶段，并将各个阶段的输出连接起来。来自一个阶段的输出被用于规划后续过程活动的基础。

图 3-1 描述了计划驱动的方法以及敏捷方法在系统规格说明方面的本质区别。在一个计划驱动的软件开发过程中，迭代发生在活动中，伴随着用于在各个过程阶段间进行交流的正式文档。例如，需求会发生演化，最终会产生一个需求规格说明。接着这个需求规格说明作为输入提供给设计和实现过程。在一个敏捷方法中，迭代发生在活动之间。因此，需求和设计会被一起开发而不是分别开发。

在实践中，如 3.4.1 节所述，计划驱动的过程经常与敏捷编程实践一起使用，敏捷方法除了编程和测试之外还可以融入一些事先计划好的活动。在一个计划驱动的过程中，按照一系列增量分配需求并计划设计和实现阶段也是完全可行的。一个敏捷过程不一定只关注代码，它也可以产生一些设计文档。敏捷开发者可能会选择在某个迭代中不产生新的代码而是产生系统模型和文档。

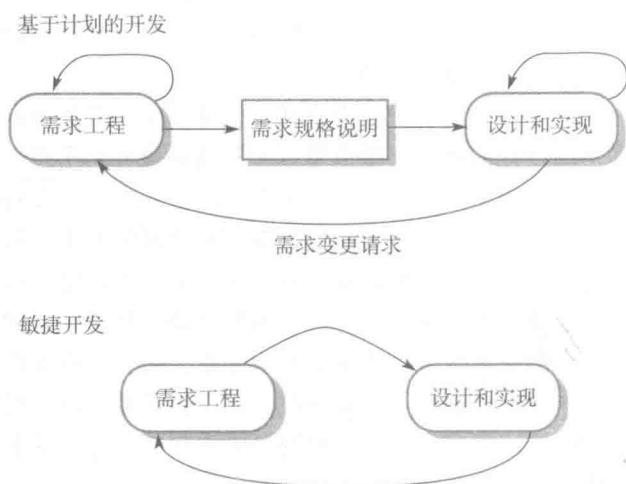


图 3-1 计划驱动的开发和敏捷开发

3.1 敏捷方法

在 20 世纪 80 年代以及 20 世纪 90 年代早期，有一种广泛传播的观点认为，获得更好的软件的最佳方法是通过仔细的项目规划、正式的质量保障、使用软件工具支持的分析和设计方法，以及受控和严格的软件开发过程。这种观点来自于负责开发大型、长期存在的软件系统（例如，航空航天系统以及政府系统）的软件工程群体。

这种计划驱动的方法针对的是由分属不同公司的大型团队开发的软件。这种团队经常在地理位置上很分散，并且长期在某个软件项目中工作。这类软件的一个例子是现代飞机的控制系统，该系统从最初的规格说明到部署可能要花费 10 年的时间。计划驱动的方法在系统的计划、设计和文档化方面有着巨大的额外开销。这些额外开销在有些情况下是合理的，例如，多个开发团队之间的工作需要协调，所开发的系统是一个关键性系统，许多不同的人会在软件整个生命期中参与软件的维护。

然而,当这种重载的计划驱动的开发方法被应用于中小规模的业务系统时,其所产生的额外开销过大,以至于占据了软件开发过程的主要部分——更多的时间花在了考虑系统应该如何开发而不是程序开发和测试本身。随着系统需求的变化,返工是必不可少的,从原则上讲至少规格说明和设计必须和程序一起修改。

对于这些重载的软件工程方法的不满导致了敏捷方法在 20 世纪 90 年代后期发展起来。敏捷方法允许开发团队关注软件本身而不是它的设计和文档化。它们最适合于应用开发,其中系统需求通常会在开发过程中快速变化。这些方法的目的是快速向客户交付可工作的软件,这样他们就可以提出系统的后续迭代中所要包含的新的需求以及需求变更。这些方法努力通过避免进行一些长期价值可疑的工作以及去除很有可能再也不会被使用的文档,以便减少过程中的“官僚主义”。

敏捷方法背后的哲学在由方法主要提出者所发表的敏捷宣言(<http://agilemanifesto.org>)中体现出来。这个宣言声称:

我们通过身体力行和帮助他人来揭示更好的软件开发方式。通过这些工作,我们形成了如下价值观:

个体与交互胜过过程和工具;

可工作的软件胜过全面的文档;

客户协作胜过合同谈判;

响应变化胜过遵循计划。

也就是说,虽然在每项对比中后者也有价值,但我们更看重前者的价值^①。

原 则	描 述
客户参与	客户应当在整个开发过程中紧密参与。他们的角色是提供新的系统需求及其优先级,并对系统的迭代进行评价
拥抱变化	期待系统需求变化,对系统进行设计以更好地融入这些变化
增量交付	软件按照增量进行开发,客户描述每个增量中要包含的需求
保持简洁	在所开发的软件以及开发过程中都关注简洁。只要有可能,都要积极地消除系统的复杂性
人而不是过程	开发团队的技能应当被充分认识和利用。应当让团队成员在没有规定的过程的限制下发展他们自己的工作方式

图 3-2 敏捷方法的原则

所有的敏捷方法都提倡软件应当以增量的方式开发和交付。这些方法基于不同的敏捷过程,但是都遵循敏捷宣言中的那些原则(图 3-2 列出了这些原则),因此不同的方法之间有很多共性。

敏捷方法已经在很多领域取得了成功,特别是在以下两类系统的开发中。

1. 软件企业所开发的用于市场销售的中小规模产品。事实上几乎所有软件产品和应用程序现在都在使用敏捷方法开发。

2. 组织内的定制化系统开发,其中客户承诺可以参与开发过程,并且影响软件开发的外部利益相关者和法规不多。

敏捷方法在这些情况下效果很好,这是因为产品经理或系统客户与开发团队之间可以进行持续的交流和沟通。软件自身是一个独立的系统而不是与其他同一时间开发的系统紧密集

① <http://agilemanifesto.org/>

成。因此，这些系统的开发不需要在多个并行的开发流之间进行协调。中小规模系统的开发团队可以都位于一个地点，因此团队成员之间的非正式交流与沟通可以进行得很好。

3.2 敏捷开发技术

敏捷方法的基本思想是在 20 世纪 90 年代由一些人大致在同一时间发展起来的。然而，其中最重要的、改变了软件开发文化的方法也许是极限编程（Extreme Programming，XP）的发展。这个名字是由 Kent Beck（Beck 1998）发明的，取这个名字的原因是该方法将得到认同的最佳实践（例如，迭代式开发）推动到“极限”的水平。例如，在 XP 方法中，一个系统的多个不同的新版本可以由不同的程序员在一天之中开发、集成和测试。图 3-3 描述了 XP 方法产生一个所开发系统的一个增量的过程。

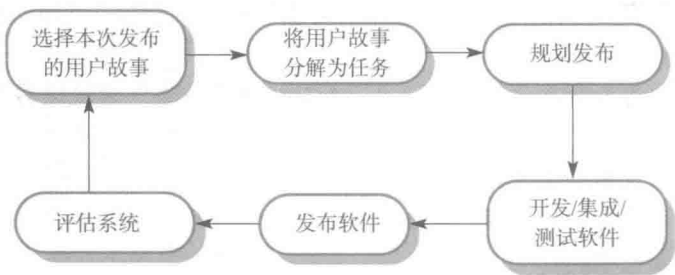


图 3-3 XP 方法的发布周期

在 XP 方法中，需求被表达为场景（称为用户故事），并作为一系列任务来直接实现。程序员结对工作，在编写代码前为每个任务开发测试。当新的代码集成到系统中时，所有的测试都必须成功执行。系统的发布之间时间间隔很短。

原则或实践	描 述
共同拥有权	开发者成对完成系统各个部分的工作，从而不会产生各部分开发知识的孤岛效应，所有的开发者对所有的代码负责。任何人都可以修改任何东西
持续集成	只要一个任务的工作完成，就将其集成到整个系统中。任何一次集成之后，系统的所有单元测试都必须通过
增量规划	需求被记录在“故事卡片”上，一个发布中将要包含的故事取决于可用的时间以及它们的相对优先级。开发者将这些故事分解为开发“任务”。见图 3-5 和图 3-6
现场客户	系统最终用户（客户）的一个代表应当全时服务于 XP 团队。在极限编程过程中，该客户是开发团队的成员，负责将系统需求带到团队中进行实现
结对编程	开发者结对工作，相互检查对方的工作并提供支持以确保总是能高质量地完成工作
重构	希望所有的开发者一旦发现潜在的代码改进机会都能持续对代码进行重构。这样可以使代码保持简洁并具有好的可维护性
简单设计	进行足够的设计以满足当前的需求，但不需要过多
小的发布	首先开发一个可以提供业务价值的最小的有用功能集合。系统频繁发布并且在第一个发布基础上增量地增加功能
可持续的步调	不接受大量的加班，因为其最终结果经常是降低代码质量和中期生产率
测试先行的开发	在一个新功能自身实现之前，使用自动化单元测试框架来为该功能编写测试

图 3-4 极限编程实践

极限编程存在争议，因为它引入了一些与当时的开发实践很不一样的敏捷实践。图 3-4

中对这些实践进行了总结，它们反映了以下这些敏捷宣言的原则。

1. 通过频繁的小的系统发布支持“增量交付”。需求是基于简单的客户故事或场景，基于这些信息开发者决定在一个系统增量中包含哪些功能。

2. 通过在开发团队中持续地约见客户实现“客户参与”。客户代表参与开发并负责定义系统的验收测试。

3. 通过结对编程、系统代码的共同拥有权，以及不包含超长工作时间的可持续的开发过程来支持“人而不是过程”。

4. 通过定期为客户提供系统发布、测试先行的开发、避免代码退化的重构、新功能的持续集成来“拥抱变化”。

5. 通过经常性的提高代码质量的重构、使用不包含不必要的系统未来变化预测的简单设计来实现“保持简洁”。

在实践中，按照最初的设想应用极限编程已经被证明比所预期的要更困难。极限编程无法与大多数企业的管理实践和文化顺利地集成。因此，采用敏捷方法的企业会选取一些最适合他们工作方式的极限编程实践。有时候这些实践会被融入他们自己的开发过程中，但更常见的情况是，将它们与关注管理的敏捷方法（例如 Scrum 方法）一起使用（Rubin 2013）。

我个人觉得对于大多数企业而言，XP 方法自身难以作为一种实用的敏捷方法，但是它最突出的贡献可能是它所引入的那一组敏捷开发实践。本节将介绍这些实践中最重要的部分。

3.2.1 用户故事

软件需求总会发生变化。为了应对这些变化，敏捷方法没有一个独立的需求工程活动，而是将需求抽取与开发集成到一起。为了使之更容易，这些方法提出了“用户故事”的思想，其中每个用户故事是一个系统用户可能经历的使用场景。

系统客户应当尽可能地与开发团队紧密工作，并且与其他团队成员一起讨论这些场景。他们一起开发一种“故事卡片”，其中简要描述了一个包含客户需要的故事。接着开发团队计划在软件的后面某个发布中实现该场景。作为例子，图 3-5 描述了 Mentcare 系统的一个故事卡片。这是一个给病人开药物处方的场景的简短描述。

开药物处方
凯特是一个想给来诊所看病的病人开药物处方的医生。病人记录已经在她的电脑上显示着，因此她可以点击其中的药物字段，并且可以选择“当前药物”“新的药物”或者“处方集”。
如果她选择“当前药物”，那么系统将请她检查剂量；如果她想改变剂量，那么可以输入新的剂量然后确认处方。
如果她选择“新的药物”，那么系统假设她知道要开什么药。她输入药物名称的前几个字母。系统显示由这几个字母开头的可能的药物列表。她选择所需要的药物，系统进行响应并请她检查所选择的药物是否正确。然后她输入剂量并确认处方。
如果她选择“处方集”，那么系统显示一个搜索框用于搜索经批准的处方集。接着她可以搜索所需要的药物。她选择一个药物然后系统请她检查药物是否正确。接着她输入剂量并确认处方。
系统总是检查药物剂量是否在所批准的范围内。如果超出范围，将会要求凯特修改剂量。
凯特确认处方后，系统会显示处方从而让她进行确认。她可以点击“确定”或“修改”按钮。如果她选择“确定”，那么该处方将会被记录在审计数据库中。如果她点击“修改”，那么她重新进入“开药物处方”过程。

图 3-5 一个“开药物处方”的用户故事

用户故事可以用于规划系统迭代。一旦故事卡片已经开发好了，开发团队将其分解为任务（见图 3-6）并且预计实现每个任务所需的工作量和资源。这通常包括与客户讨论以精化需求。接着，客户确定这些故事的实现优先级，选取那些可以立即使用以提供有用的业务支持的故事。其目的是识别可以在大约两周之内实现的有用的功能，到时候系统的下一个发布可以被交付给客户。

当然，如果需求发生变化，那么未实现的故事会发生变化或者被抛弃。如果某个已经交付的系统需要变更，那么需要开发新的故事卡片，并且客户再次决定这些变化是否应当比新功能具有更高的优先级。

用户故事的思想很强大，人们发现与传统的文档或用况相比，使用这些故事要容易得多。用户故事在让用户参与初始开发之前的需求抽取活动并提出需求等方面很有帮助。第 4 章将详细介绍这一点。

用户故事的主要问题是完整性。很难判断是否已经开发了足够的用户故事以覆盖一个系统所有的重要需求。判断单个故事是否正确描述了一个活动也不容易。有经验的用户经常很熟悉自己的工作，以至于会在描述时遗漏一些东西。

3.2.2 重构

传统软件工程的一个基本原则是开发者应该在设计中考虑未来的变更。也就是说，应该预测未来针对软件的变化并在设计中考加以考虑，从而使这些变化可以很容易实现。然而，极限编程摒弃了这一原则，其依据是面向变化的设计经常是多余的和无用的。不值得花时间在程序中增加通用性以应对变化。所预计的变化经常并不会发生，或者实际发生的变更请求与所预计的变更完全不同。

当然，在实践中，所开发的代码中总是会发生变化。为了使这些变化更容易实现，极限编程方法建议开发者不断对所开发的代码进行重构。重构（Fowler et al. 1999）的意思是编程团队寻找对软件进行改进的可能性并立即实现这种改进。当团队成员看到可以改进的代码，他们马上实施改进，即使这一改进对于他们来说并不是非常紧迫。

增量开发的一个基本问题是局部的修改可能会导致软件结构的退化。其结果是，对于软件的进一步修改变得越来越难以实现。本质上讲，开发过程就是在寻找问题的替代方案的过程中前进的，其结果是代码经常发生重复、软件的某些部分以不恰当的方式被复用，以及整体结构随着代码不断被加入系统而发生退化。重构改进软件的结构和可读性，因此可以避免当软件被修改时自然发生的结构退化。

重构的例子包括对一个类的继承层次进行重新组织，以便去除重复的代码、对属性和方法进行整理和重命名、用对程序库中定义的方法的调用来代替相似的代码段。程序开发环境

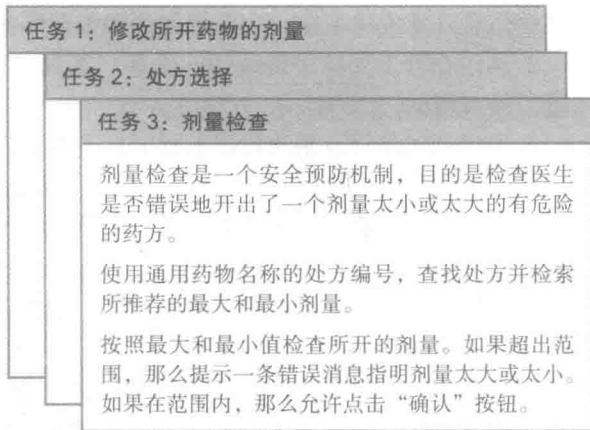


图 3-6 “开药物处方”的任务卡片例子

通常包含支持重构的工具。这些工具简化了寻找代码段之间依赖关系以及进行全局的代码修改的过程。

原则上讲,当重构成为开发过程的一部分,软件在新需求被提出时应当总是很容易理解和修改的。然而在实践中并不总是如此。有时候开发压力意味着重构会被推迟,因为时间都被用于实现新功能。一些新的特征和变更无法通过代码级的重构顺利地融入进来,需要对系统的体系结构进行修改。

3.2.3 测试先行的开发

如本章的引言部分所述,增量的开发和计划驱动的开发的一个重要区别是系统测试的方式。使用增量开发时,没有供外部的测试团队使用的用于开发系统测试的系统规格说明。其结果是,与计划驱动的测试相比,一些增量开发方法所采用的测试过程很不正式。

极限编程开发了一种新的程序测试方法以应对在没有规格说明的情况下测试所面临的困难。测试被自动化了而且处于开发过程的核心地位,只有在所有测试都已经成功执行之后开发过程才能继续进行。极限编程方法中测试的关键特性如下。

1. 测试先行的开发;
2. 基于场景的增量测试开发;
3. 用户参与测试开发和确认;
4. 使用自动化测试框架。

极限编程方法的测试先行哲学现在已经逐渐演化成了更加一般化的测试驱动的开发技术(Jeffries and Melnik 2007)。我认为测试驱动的开发是软件工程中最重要创新之一。与传统的先写代码然后为这些代码编写测试的方式不同,开发人员在写代码之前先编写测试。这意味着开发人员可以在代码编写过程中运行测试,并且在开发中发现问题。第8章将更深入地介绍测试驱动的开发。

编写测试隐式地为所开发的功能定义了接口以及行为规约,这使得需求中的问题以及接口误解的问题都可以减少。测试优先的开发要求系统需求与实现对应需求的代码之间存在清晰的关系。在极限编程中,这种关系很清楚,因为表示需求的故事卡片被分解为任务,而任务是主要的实现单元。

在测试先行的开发中,任务的实现者必须透彻理解规格说明,从而能够为系统编写测试。这意味着规格说明中的需求歧义和遗漏在实现开始之前必须被澄清。此外,这也可以避免“测试拖后腿”的问题。这一问题会在系统的开发人员比测试人员的工作进度快的情况下发生。如果实现进行到测试之前很远很远的地方了,那么可能会产生跳过测试的倾向,从而使开发进度得以保持。

极限编程的测试先行的方法假设用户故事已经开发好,并已经被分解为一组任务卡片,如图3-6所示。每个任务产生一个或多个单元测试,以便检查任务中所描述的实现。图3-7是一个测试用例的部分描述,开发这个测试用例的目的是检查所开的药品剂量没有超出已知的安全范围。

客户在测试过程中的角色是帮助开发将在系统下一个发布中实现的用户故事的验收测试。如第8章中所述,验收测试使用客户数据测试系统以检查系统是否满足客户的真实需要。

测试 4: 剂量检查
输入: 1. 一个表示单次服药剂量的数字 (单位: 毫克) 2. 一个表示每天服药次数的数字
测试: 1. 针对输入进行测试, 检查是否单次剂量正确但频率太高。 2. 针对输入进行测试, 检查是否单次剂量太大或太小。 3. 针对输入进行测试, 检查单次剂量 \times 频率是否太大或太小。 4. 针对输入进行测试, 检查单次剂量 \times 频率是否在所允许的范围内。
输出: 确认无误或者提示错误信息表明剂量超出安全范围。

图 3-7 剂量检查测试用例描述

测试自动化对于测试先行的开发很重要。测试在任务实现之前被编写为可运行的构件。这些测试构件应当是独立的, 应当模拟向被测试代码提交输入, 应当检查运行结果是否满足输出规约。自动化测试框架是一种使人们很容易地编写可运行的测试以及提交一组测试进行运行的系统。例如, Junit (Tahchiev et al. 2010) 是一种广泛使用的 Java 程序自动化测试框架。

由于测试被自动化了, 因此总是有一组可以很快并且很容易执行的测试。每当任何一个新的功能被加入系统中时, 测试可以运行, 而新代码所引入的问题可以立即被捕捉到。

测试先行的开发以及自动化测试通常导致大量的测试被编写和执行。然而, 在保证测试覆盖度的完备性方面还存在以下这些问题。

1. 程序员更喜欢编程而不是测试, 有时候他们会在编写测试时走捷径。例如, 他们可能会编写无法检查所有可能发生的执行异常的不完整测试。

2. 有些测试很难增量地编写。例如, 在一个复杂的用户界面中, 为实现“显示逻辑”以及屏幕之间的工作流的代码编写单元测试经常会很困难。

判断一组测试的完备性是很难的。虽然可以有系统测试, 但是测试集合可能并没有提供完备的覆盖度。系统的一些关键部分可能没有执行到, 因而没有被测试到。因此, 虽然一个很大的且频繁被执行的测试集合可能会给人一种系统很完备、很正确的印象, 但是事实可能并非如此。如果测试没有经过评审而进一步的测试是在开发之后编写的, 那么未发现的缺陷可能会留在系统发布中被交付出去。

3.2.4 结对编程

极限编程所引入的另一个创新性的实践是程序员结对一起工作来开发软件。结对的程序员坐在一起用同一台电脑来开发软件。然而, 同一对程序员并不总是一起编程。结对的程序员是动态确定的, 从而使所有的团队成员都能在开发过程中一起工作。

结对编程的好处包括以下这些方面。

1. 支持对于系统的共同所有权和共同责任的思想。这反映了 Weinberg 的“无私编程”的思想 (Weinberg 1971), 其中软件由团队整体所有, 个人不对代码中的问题负责。团队对于这些问题的解决负有共同责任。

2. 扮演了非正式的评审过程的角色, 因为每一行代码都由至少两个人看着。代码审查和

评审（第24章）可以有效地发现很大一部分软件错误。然而，代码审查和评审的组织非常耗时，经常导致开发进度延迟。结对编程没有那么正式，可能无法像代码审查那样找到很多错误。但是，与正式的程序审查相比，组织结对编程的成本更低也更容易。

3. 鼓励通过重构改进软件的结构。让程序员在常规的开发环境中进行重构的一个问题是，所花费的工作量是为了获取长期的收益。有些人认为，相比那些不做重构而只是简单地开发代码的开发者，花时间进行重构的开发者的开发效率可能要低一些。当使用结对编程和共同的所有权时，其他人可以立即从重构中获益，因此他们会更加支持重构。

你可能会认为结对编程没有各自独立编程效率那么高。在给定的时间内，一对开发者产出的代码量可能只有两个开发者各自独立开发的一半那么多。许多已经采取敏捷方法的企业怀疑结对编程并且没有使用。其他一些企业则将结对编程与各自独立编程混合在一起，在发生问题时让一名经验丰富的程序员与一名经验没那么丰富的同事一起工作。

对于结对编程的价值的正式研究产生了多种不同的结果。Williams和她的合作者（Williams et al. 2000）使用学生志愿者进行研究，发现结对编程的生产率与两个人各自独立工作差不多。其原因被认为是一对开发者可以在开发之前对软件进行讨论，因此很有可能在开始时的误解会更少而返工也更少。而且，有很多错误都可以通过非正式的审查来避免，从而使花费在修复测试过程中发现的缺陷上的时间大大减少。

然而，针对更有经验的程序员的研究没有重复这些结果（Arisholm et al. 2007）。他们发现与两个程序员各自独立工作相比，结对编程在生产率上的损失很显著。在代码质量上会有一些好处，但是这并不能完全补偿结对编程所带来的额外开销。然而，在结对编程过程中所发生的知识的共享很重要，因为这样可以降低由于团队成员离开而带来的项目风险。这一点可能使结对编程值得采用。

3.3 敏捷项目管理

在任何软件企业中，管理人员都需要了解项目进展情况如何，以及项目是否能满足其目标并在预算范围内按时交付软件。计划驱动的软件开发方法不断发展以满足这一要求。如第23章中所述，管理人员制订一个项目计划，其中显示要交付什么东西、什么时候要交付、谁应该负责项目交付物的开发。计划驱动的方法要求一个管理者对于要开发的一切以及开发过程都有一个稳定的视图。

敏捷方法的早期追随者所提出的非正式的计划和项目控制与这一针对可见性的业务需求相冲突。团队是自组织的，不会产生文档，所计划的开发都是很短的周期。虽然这些对于开发软件产品的小公司而言能够并且也的确可行，但是对于需要知道组织中所发生的事情的大公司而言就不适合了。

与其他任何一个专业化的软件开发过程一样，敏捷开发必须处于有效的管理之中，以使团队所获得的时间和资源能够得到充分利用。为了应对这一问题，Scrum敏捷方法（Schwaber and Beedle 2001; Rubin 2013）被提出以提供一个组织敏捷项目的框架，从而至少在一定程度上提供项目进展状况的外部可见性。Scrum的提出者希望明确的一点是，Scrum不是一种通常意义上的项目管理方法，因此他们革命性地发明了一些新的术语，例如，Scrum主管（ScrumMaster）取代了项目经理等原有的名称。图3-8总结了Scrum中的一些术语及其含义。

术 语	定 义
开发团队	一个自组织的软件开发者小组，不超过 7 人。他们负责开发软件以及其他重要的项目文档
潜在的可交付的产品增量	通过一个冲刺（sprint）交付的软件增量。其思想是该增量应该是“潜在可交付的”，这意味着该增量处于已完成状态，并且不需要进一步的工作（例如测试）来将其集成到最终产品中。在实践中，这一点并不总是可以实现的
产品代办事项	这是一个 Scrum 团队必须处理的待办事项的列表。它们可以是软件的特征定义、软件需求、用户故事，或者所需的补充任务（例如体系结构定义或者用户文档）的描述
产品负责人	一个人（或一个小组），其任务是识别产品特征或需求，确定它们的开发优先级，并持续地评审产品待办事项以确保项目持续满足关键的业务要求。产品负责人可以是一个客户，但也可能是一个软件公司的产品经理或者其他利益相关者的代表
每日站立会议	Scrum 团队每天的例行会议，用于对进度进行评审，并且对当天要进行的工作进行优先级排序。理想情况下，该会议应该是一个整个团队都参加的简短的面对面会议
Scrum 主管	Scrum 主管负责确保 Scrum 过程得到遵循，并且指导团队有效地使用 Scrum。他负责与公司其他部分的接口，并确保 Scrum 团队不会受到外部的干扰。Scrum 开发者可以坚持己见，Scrum 主管不应该被认为是项目经理。然而其他人并不总能轻易地看到其中的区别
冲刺	一种开发迭代。冲刺通常长达 2 ~ 4 周
速度	对一个团队在单个冲刺中可以完成多少产品待办事项工作量的预测。理解一个团队的速度可以帮助他们预测一次冲刺中可以完成多少事情，并为衡量团队开发表现的改进提供了依据

图 3-8 Scrum 术语表

Scrum 是一种敏捷方法，因为它遵循了敏捷宣言中的那些原则（见图 3-2）。然而，该方法关注为敏捷项目组织提供一个框架，并没有指定使用一些特定的开发实践，例如，结对编程、测试先行的开发等。这意味着在一个企业中 Scrum 可以更容易地与已有的实践相集成。因此，由于敏捷方法已经成为软件开发的主流方法，Scrum 也成为最广泛使用的方法。

Scrum 过程，或称为冲刺周期，如图 3-9 所示。过程的输入是产品待办事项。每次过程迭代都会产生一个可以交付给客户的产品增量。

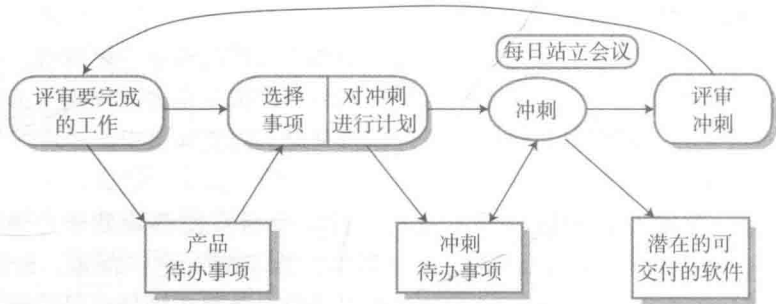


图 3-9 Scrum 冲刺周期

Scrum 冲刺周期的起点是产品待办事项——Scrum 团队必须完成的产品特征、需求和工程改进等事项的列表。产品待办事项的初始版本可以从需求文档、一个用户故事列表或者其他待开发软件的描述中派生出来。

虽然产品待办事项中大部分条目都关注系统特征的实现,但是也有可能包含其他活动。有时候,当计划一次迭代时可能会发现一些很难回答的问题,因此需要进行一些额外的工作来探索可能的解决方案。团队可以进行一些原型或试验性的开发活动以更好地理解问题和解决方案。还可能在设计系统体系结构或开发系统文档的待办事项。

产品待办事项的详细程度可以各不相同,产品负责人负责确保其中的详细程度与要完成的工作相适应。例如,一个待办事项可以是一个完整的用户故事(如图3-5所示),或者只是一个简单的指令,例如,只是说明“重构用户界面代码”而让开发团队决定要进行哪些重构。

每个冲刺周期持续一个固定的时长,通常是2~4周。在每个周期开始时,产品负责人对产品待办事项上的各项内容进行优先级排序,以定义该周期中要开发的最重要的项是哪些。冲刺从来不会为了考虑未完成的工作而进行扩展。在当前冲刺周期的时间内无法完成的项会被返回给产品待办事项。

接下来,整个团队参加选取他们认为可以完成的最高优先级的事项。他们接着预测完成这些事项所需的时间。为了进行预测,他们使用从此前的冲刺中所获得的速度,即单个冲刺中可以完成多少待办事项。这导致了冲刺待办事项的创建,即当前冲刺期间要完成的工作。团队通过自组织的方式确定各部分工作任务的分配,接着冲刺开始。

在冲刺过程中,团队每天进行简短的会议(每日站立会议)来对进度进行评审,并在必要时重新对工作进行优先级排序。在每日站立会议中,所有的团队成员共享信息,描述他们自上一次会议之后的进度,汇总所发现的问题,并陈述当天计划完成的事情。这样,团队中的每一个成员都知道进展情况,并且在发生问题时重新计划短期工作以应对问题。每个人都参与这一短期计划过程;没有自顶向下来自Scrum主管的指示。

Scrum团队成员之间每天的交互可以使用一个Scrum板进行协调。这是一个办公室里的白板,其中包括关于冲刺待办事项、已完成的工作、缺席人员等的信息和便利贴。这是整个团队的共享资源,每个人都可以改变或移动板上的东西。这意味着任何团队成员都可以一下子看到其他人正在做什么以及还有哪些工作待完成。

在每个冲刺结束的时候,有一个整个团队参加的评审会议。这个会议有两个目的:第一,这是一种过程改进的手段。团队成员对他们此前的工作方式进行评审,并反思事情是否可以做得更好。第二,为下一个冲刺之前的产品待办事项评审提供了关于产品和产品状态的输入。

虽然Scrum主管并不是正式的项目经理,但是在实践中许多采用传统管理结构的组织的Scrum主管扮演着项目经理的角色。他们向高层管理报告进度,并参与长期的计划和项目预算。他们可以参与项目行政管理(例如,同意员工休假、联系人事部门等)以及硬件和软件采购。

在各种各样的Scrum成功故事中(Schatz and Abdelshafi 2005; Mulder and van Vliet 2008; Bellouiti 2009),用户喜欢Scrum方法中以下几个方面。

1. 产品被分解为一组可管理、可理解、利益相关者可以对应上的条块。
2. 不稳定的需求不会影响进度。
3. 整个团队都对所有的一切保持可见,因此团队交流沟通和士气都得到了提升。
4. 客户可以按时看到增量的交付,并获得关于产品如何工作的反馈。他们不会在最后一刻当开发团队宣布软件无法按时交付时感到惊诧。
5. 客户和开发者之间建立了信任,建立了积极的文化,其中每个人都期望项目能成功。

按照最初的设想, Scrum (站立式会议) 将被用于同处一地的团队, 其中所有的团队成员都可以每天在一起参加站立式会议。然而, 很多软件开发项目现在都包含分布式的团队, 其中团队成员位于世界上不同的位置。这使得企业可以利用其他国家中成本较低的员工, 可以得到所需的专家技能, 并且使 24 小时开发成为可能 (在不同时区进行的工作)。

因此, 现在 Scrum 方法已经针对分布式开发环境和多团队工作进行改进。通常, 对于离岸开发, 产品负责人与开发团队位于不同的国家, 而开发团队本身也有可能是分布式的。图 3-10 显示了分布式 Scrum 方法 (Deemer 2011) 的要求。

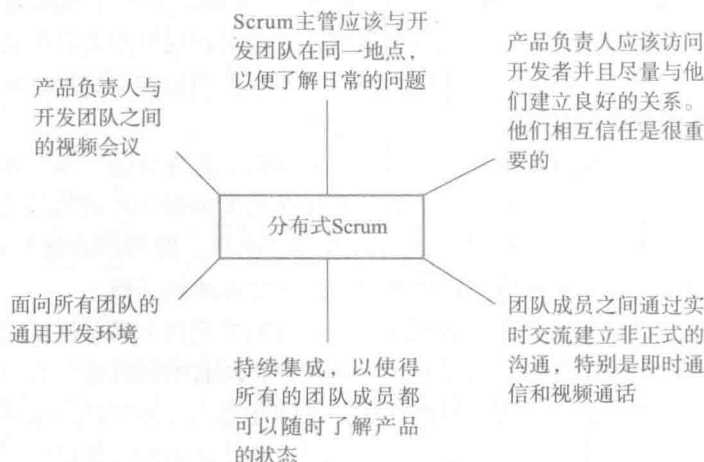


图 3-10 分布式 Scrum

3.4 敏捷方法的伸缩

敏捷方法最初被开发时, 其目的是服务于可以在同一个房间一起工作并进行非正式的交流的小编程团队。最初人们利用这些方法开发中小规模的系统和软件产品。没有正式的过程和官僚主义的小公司是这些方法最初的热情支持者和使用者。

当然, 更快地交付软件 (这更适合于客户要求) 的要求同样适用于大型系统和大公司。因此, 在过去这些年中, 人们投入人力物力发展敏捷方法, 努力使其同样可以用于大型软件系统和大公司之中。

敏捷方法的伸缩包括以下两个密切相关的方面。

1. 将这些方法规模化以处理大系统的开发, 这些系统往往因为太大而无法由单个的小团队来完成。
2. 将这些方法的应用范围从专门的开发团队扩展到在一个有着多年软件开发经验的大企业内更加广泛地使用。

当然, 这里的规模化和扩展是密切相关的。开发大型软件系统的合约通常都会与较大的组织签订, 其中会有多个团队在开发项目上工作。这些大企业经常已经在较小的项目上试验过敏捷方法, 因此他们同时面临规模化和扩展的问题。

关于敏捷方法的有效性有很多趣闻, 有很多都提到敏捷方法的应用可以带来生产力方面数量级的提升以及与此相当的缺陷数量的减少。Ambler (Ambler 2010) 是一个有影响力的

敏捷方法开发者，他认为敏捷方法对于大型系统和组织的生产力提升被夸大了。他认为一个向敏捷方法转型的组织可以期望看到整个组织在3年内生产力提升15%左右，同时产品缺陷数量下降比例也差不多是15%。

3.4.1 敏捷方法的实践问题

在某些领域，特别是软件产品和应用程序的开发中，敏捷开发已经取得了令人难以置信的成功。敏捷方法是到目前为止对于这类系统最好的开发方法。然而，敏捷方法可能并不适合于其他类型的软件开发，例如嵌入式系统工程或者大型复杂系统的开发。

对于由软件公司为外部客户开发的大型、长生命周期的系统，使用敏捷方法还存在着下面这些问题。

1. 敏捷开发的非正式性与大型企业中通常使用的基于法律的定义不相符。
2. 敏捷方法最适合于新的软件开发而不是软件维护。然而，大型企业软件成本大部分来自于对于已有软件系统的维护。
3. 敏捷方法适用于小的、同处一地的团队，然而当前的很多软件开发都包含全球分布的开发团队。

当使用敏捷方法时，合同问题会是一个主要的问题。当系统客户使用外部的组织进行系统开发时，需要相互之间签订一个开发合同。软件需求文档通常会成为客户和供应商之间合同的一部分。由于敏捷方法中一个基本要点是，需求和代码的开发相互交织，因此在合同中很难对需求给出确定性的陈述。

因此，敏捷方法必须依赖于合同中明确客户按照系统开发所需要的时间，而不是完成一组特定需求的开发来支付费用。如果一切进展顺利，那么这对于客户和开发者都是有利的。然而，如果出现问题，那么很容易出现争议，很难判断哪一方应该承担责任并为解决问题所需要的额外时间和资源开销买单。

如第9章所述，大量的软件工作被用于已有软件系统的维护和演化。敏捷实践，例如增量交付、面向变化的交付、保持简洁，在软件发生变化时都是有道理的。事实上，你可以认为敏捷开发过程是一种支持持续变化的过程。如果敏捷方法用于软件产品开发，那么新的产品或应用程序发布只是敏捷方法的持续使用。

然而，当维护工作涉及一个必须按照新的业务需求进行变更的定制化系统时，敏捷方法对于软件维护是否适用并没有一个清晰的共识（Bird 2011; Kilner 2012）。可能会产生以下3类问题。

- 缺少产品文档；
- 保持客户参与；
- 开发团队的延续性。

正式的文档被认为可以描述系统，从而使得修改系统的人们可以更容易地理解系统。然而，在实践中正式的文档很少能及时更新，因此并不能准确地反映程序代码。因此，敏捷方法的拥护者们强调书写文档是浪费时间，而实现可维护的软件的关键是产出高质量、可读的代码。维护使用敏捷方法开发的系统时，缺少文档不应该是问题。

然而，根据我的系统维护经验，最重要的文档是系统需求文档，它可以告诉软件工程师用户希望所开发的系统做什么。没有这样的认识，很难对所提出的系统变更的影响进行评价。许多敏捷方法以增量的以及非正式的方式收集需求，不会创建一个清晰的需求文档。因

此，使用敏捷方法可能会使随后的系统维护更加困难、成本更加昂贵。如果开发团队的延续性无法得到保持，那么问题会更大。

使用敏捷方法进行维护的一个关键挑战是让客户参与到过程中来。虽然客户可能会觉得在系统开发过程中作为代表全时参与开发团队是合理的，但对于维护过程（变化并不是总是持续发生的）却很难这样认为。客户代表很容易对系统失去兴趣。因此，很可能需要对其他替代性的机制（例如第 25 章中介绍的变更提议）进行调整以适应敏捷方法。

另一个可能出现的潜在问题是保持开发团队的延续性。敏捷方法依赖于团队成员在没有文档的情况下理解系统的各个方面。如果敏捷开发团队解散了，那么这些隐式的知识会丢失，新的团队成员很难对系统及其各个构件构建起同样的理解。许多程序员更喜欢进行新的开发工作而不是进行软件维护，因此他们会不愿意在一个软件系统的第一个发布版本交付后继续参与该系统的开发。因此，即使有意识地将开发团队保持在一起，如果开发人员被分配去做维护任务，他们也会选择离开。

3.4.2 敏捷和计划驱动的方法

规模化和扩展敏捷方法的一个根本要求是与计划驱动的方法相集成。小的初创企业可以使用非正式的短期计划，但是更大的大企业必须有更长期的计划以及与投资、人员、业务发展相关的预算。他们的软件开发必须支持这些计划，因此更长期的软件计划是很重要的。

21 世纪初期的那些敏捷方法的早期使用者都比较狂热，对于敏捷宣言深信不疑。他们有意将计划驱动的软件工程方法排除在外，不愿意以任何方式改变敏捷方法最初的愿景。然而，随着一些组织发现敏捷方法的价值和益处，他们对这些敏捷方法进行了调整以适应他们自己的文化和工作方式。他们必须这样做，因为敏捷方法的基本原则有时候很难在实践中实现（见图 3-11）。

原 则	实 践
客户参与	这有赖于客户愿意并且能够花时间与开发团队在一起，并且能代表所有的系统利益相关者。客户代表经常要在其他事情上花时间，无法全时参加软件开发。如果还存在外部的利益相关者，例如监管者，那么很难有人能在敏捷团队中代表他们的观点
拥抱变化	对变化进行优先级排序可能极其困难，特别是有很多利益相关者的系统。通常，每个利益相关者会给不同的变更不同的优先级
增量交付	开发的快速迭代和短期计划并不总是与业务和市场的长期计划周期相适应。市场经理可能要提前几个月了解产品特性，从而能够准备一个有效的营销活动
保持简洁	在交付进度的压力下，团队成员可能没时间进行所希望的系统简化工作
人而不是过程	某些团队成员的个性可能并不适合于敏捷方法通常所要求的热情参与，因此可能与其他团队成员的交互并不太好

图 3-11 敏捷原则和组织实践

为了应对这些问题，大多数大型的“敏捷”软件开发项目都来自计划驱动的方法与敏捷方法的实践相结合。有些几乎主要是敏捷实践，而其他一些则主要是计划驱动但带有一些敏捷实践。为了确定如何在基于计划的方法和敏捷方法之间做出平衡，你必须回答一系列与技术、人和组织相关的问题。这些与所开发的系统、开发团队、开发和采购系统的组织相关（见图 3-12）。

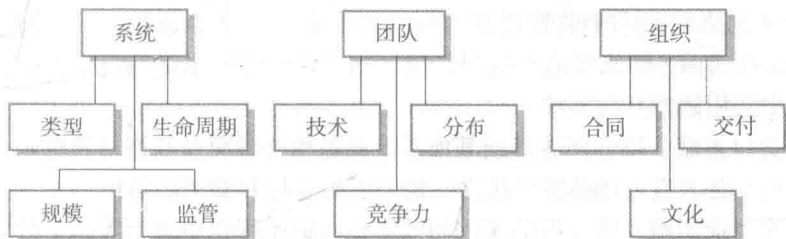


图 3-12 影响基于计划或敏捷开发方法选择的因素

敏捷方法被开发以及项目中不断细化，以便用于开发中小规模的业务系统和软件产品，其中软件开发者控制着系统的规格说明。其他类型的系统在规模、复杂性、实时响应、外部监管等方面所具有的特性意味着“纯粹的”敏捷方法很难奏效。这些系统需要在系统工程过程中提前进行计划、设计和文档化。其中的一些关键问题如下。

1. 所开发的系统有多大？敏捷方法在系统能够由一个相对较小且处于同一地点、可以进行非正式交流的团队开发时最有效。这对于需要更大的开发团队的大型系统而言可能不太可能，因此必须使用计划驱动的方法。

2. 所开发的系统是什么类型？需要在实现之前做很多分析的系统（例如，具有复杂的时间性需求的实时系统），通常需要一个相当详细的设计来进行这种分析。计划驱动的方法在这种环境下可能是最佳选择。

3. 所期望的系统生命周期有多长？长生命周期的系统可能需要更多的设计文档化，从而将系统开发者最初的意图传达给支持团队。然而，敏捷方法的支持者坚持认为，文档经常无法保持最新并且对于长期的系统维护没什么用，在这一点上他们是正确的。

4. 系统是否要接受外部监管？如果一个系统必须得到外部监管者的批准（例如，联邦航空管理机构负责审批影响飞机安全运行的关键软件），那么很可能必须提供详细的文档以作为系统安全用例的一部分。

敏捷方法对开发团队在系统开发过程中进行协作和沟通施加了很多责任。他们的开发过程依赖于个人的工程技能和软件支持。然而，在现实中不是每个人都是一个高技能的工程师，人们可能会无法有效交流，而团队也不总是能够在一起工作。可能需要一些计划来充分利用可用的人。关键问题如下。

1. 开发团队中的设计者和程序员怎么样？有人认为敏捷方法所需要的技能水平比基于计划的方法要求更高。在基于计划的方法中，程序员只需要将详细的设计转换为代码。如果开发团队技能水平相对较低，那么可能要用最好的人来开发系统设计，而让其他人负责编程。

2. 开发团队是如何组织的？如果开发团队分布在不同地方或者一部分开发任务是外包的，那么可能需要开发设计文档，利用该文档进行跨开发团队的交流和沟通。

3. 可用的支持系统开发的技术有哪些？敏捷方法经常依赖于好的工具来对不断演化的设计进行追踪。如果开发者在一个不具备好的程序可视化和分析工具的集成开发环境下开发一个系统，那么可能需要更多的设计文档。

电视剧和电影已经造成了一种被广泛接受的关于软件公司的印象，即由年轻人（大部分）管理，提供流行时尚的工作环境，几乎没有官僚主义和组织规程。事实远非如此。大多数软件都是在已经建立了自己的工作实践和规程的大企业里开发的。这些企业中的管理人员可能会对敏捷方法中缺少文档以及非正式的决策等做法感到不适。其中的关键问题如下。

1. 在开始实现之前有一个非常详细的规格说明和设计很重要吗（也许是出于合同的原因）？如果是这样的话，那么很有可能要使用计划驱动的方法来进行需求工程，但是可以在系统实现过程中使用敏捷开发实践。

2. 增量的交付策略，即你向客户或其他系统利益相关者交付软件并获得来自他们的快速反馈，这现实吗？是否有可用的客户代表，他们愿意参加开发团队吗？

3. 是否存在可能影响系统开发的文化问题？传统的工程化组织有着基于计划的开发的文化，因为这是工程化的一般做法。这通常需要全面的设计文档而不是敏捷过程中所使用的非正式的知识。

在现实中，一个项目是否可以贴上计划驱动或敏捷开发的标签并不那么重要。最终，软件系统的购买者的主要关注点是他们能否获得满足他们需要、为各个用户或组织做有用的事情、可运行的软件系统。软件开发应当采取实用主义的态度，无论他们被贴上敏捷或计划驱动的标签，都应该选择那些对于所开发的系统类型最有效的方法。

3.4.3 面向大型系统的敏捷方法

敏捷方法必须发展变化以适用大规模软件开发。其中的根本原因是大规模软件系统的理解和管理比小规模系统或软件产品要复杂和困难得多。这种复杂性来源于6个主要的因素（见图3-13）。

1. 大型系统通常都是系统之系统，即独立的且相互通信的系统集合，其中不同的独立团队分别开发每个系统。这些团队经常处在不同的地方工作，有时候还会分布于不同的时区。而每个团队想全面了解整个系统则尤其困难。因此，他们通常都会选择在不考虑更广范围内的系统问题的情况下完成他们自己所负责的那部分系统。

2. 大型系统是棕地（brownfield）系统（Hopkins and Jenkins 2008），即包含一些现有的系统并且要与之交互。系统需求中很大一部分都关注这种交互，因此难以真正地具有灵活性并适合增量开发。这里的政治问题也可能会很突出，一个最容易的解决方案经常是修改一个已有系统。然而，这需要与该系统的管理人员协商，以说服他们所进行的修改可以在不给系统运行带来风险的情况下实现。

3. 当通过集成多个系统创建一个新系统时，开发工作中很大一部分都关注子系统的配置而不是原始的代码开发。这与增量开发以及频繁的系统集成的思想不太相符。

4. 大型系统及其开发过程经常受到外部的规则和监管制约，这限制了系统的开发方式，并且要求产生特定类型的系统文档等。客户可能会有一些必须遵循的特定的兼容性需求，这些可能要求提供过程文档。

5. 大型系统的采购和开发时间很长。维持一个具有延续性、了解系统的整个开发过程的团队是很困难的，因为人们不可避免地会调动到其他工作岗位或项目中。

6. 大型系统通常有一组多种多样的利益相关者，他们的观点和目标各不相同。例如，护士和行政管理人员可以是一个医疗系统的最终用户，但是高级的医护人员、医院管理者也是

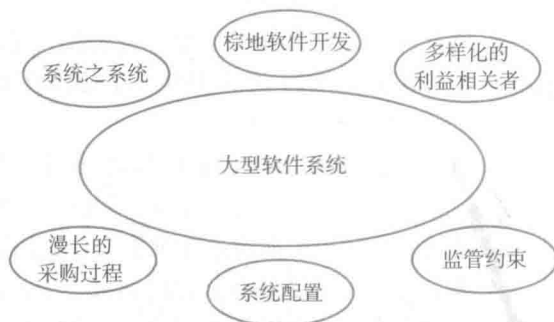


图 3-13 大型项目的特点

系统中的利益相关者。在实践中不可能把所有这些不同的利益相关者都纳入到开发过程中。

Dean Leffingwell 在敏捷方法的规模化方面有很丰富的经验，他开发了一种规模化敏捷框架 (Scaled Agile Framework)(Leffingwell 2007, 2011) 来支持大规模、多团队的软件开发，并说明了这个方法如何成功应用于一些大型企业。IBM 也开发了一个用于敏捷方法的大规模使用的框架，称为敏捷规模化模型 (Agile Scaling Model, ASM)。图 3-14 取自 Ambler 讨论 ASM 的白皮书 (Ambler 2010)，描述了这个模型的概貌。

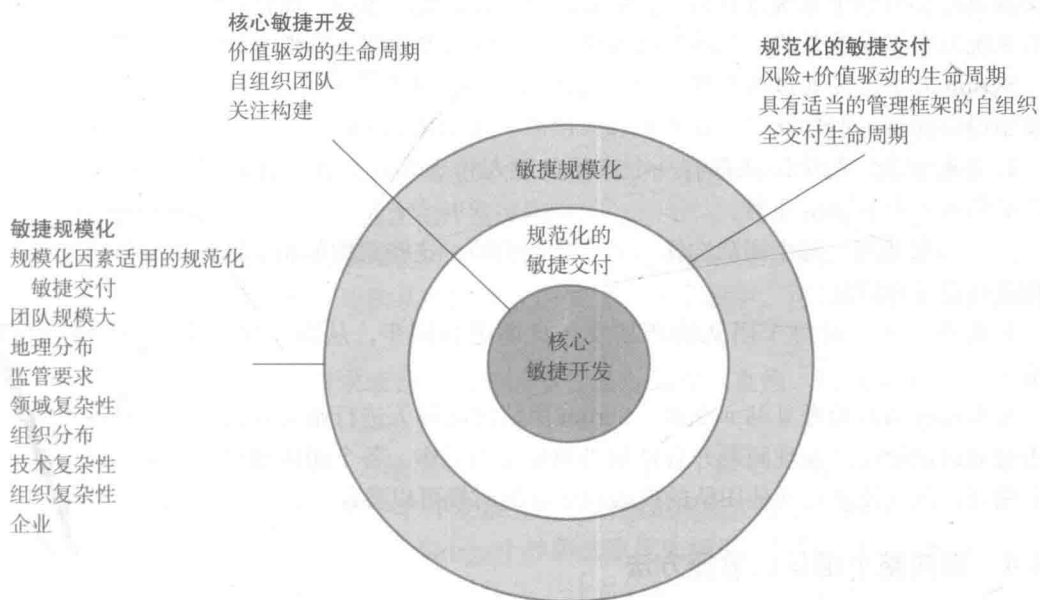


图 3-14 IBM 的敏捷规模化模型 (© IBM 2010)

ASM 认识到规模化是一个阶段性的过程，其中开发团队从这里讨论的核心敏捷实践逐渐转变为所谓的规范化的敏捷交付。从根本上讲，这一阶段包含了对于这些实践的调整以适应规范化的组织设定，并且认识到了团队不能简单地关注开发而是必须同时考虑软件工程过程的其他阶段（例如，需求和体系结构设计）。

ASM 最终的规模化阶段是进入到规模化敏捷，其中考虑到了大型项目所固有的复杂性。这包括对分布式开发、复杂的遗留环境、外部监管要求等因素的考虑。用于实现规范化敏捷交付的实践必须根据不同项目进行修改，以考虑这些因素，有时还需要在过程中增加额外的基于计划的实践。

没有哪一个模型适合于所有的大规模敏捷产品，因为产品的类型、客户需求、可用的人都不一样。然而，规模化的敏捷方法有以下这些共性的东西。

1. 一个完全增量的需求工程方法是不可能的。在初始的软件需求上进行一些早期的工作是很重要的。开发人员需要通过这个来确定系统包含哪些可以由不同团队开发的不同部分，这些经常会成为系统开发合同的一部分。然而，这些需求通常不需要详细刻画，细节最好增量地进行开发。

2. 没有哪个人能单独作为产品负责人或客户代表。不同的人必须参与系统的不同部分，他们必须在整个开发过程中持续地进行交流和协商。

3. 不可能只关注系统的代码。开发人员需要进行更多的前期设计和系统的文档化。必须

设计软件体系结构，必须产生一些描述系统关键方面的文档，例如数据库模式、团队内的工作分解。

4. 必须设计和实施整个团队的沟通机制。这应当包括团队成员之间常规的电话和视频会议，以及频繁进行的简短的电子会议以便团队成员相互更新进度。应当提供各种各样的沟通渠道，例如，电子邮件、即时通信、Wiki、社交网络系统等，以有助于成员交流和沟通。

5. 当系统必须通过多个独立程序的集成来创建时候，持续集成（其中任何一个开发人员提交修改时都对整个系统进行构建）在实践中无法实现。然而，保持频繁的系统构建以及常规的系统发布是很重要的。支持多团队软件开发的配置管理工具也十分重要。

Scrum 方法已经通过调整来适应大范围的开发。从本质上说，3.3 节中描述的 Scrum 团队模型得以保持，但是设定了多个 Scrum 团队。多团队 Scrum 方法的关键特性如下。

1. 角色重复。每个团队都有一个产品负责人和 Scrum 主管。整个项目可能还有一个首席产品负责人和 Scrum 主管。

2. 产品架构师。每个团队选择一个产品架构师，这些架构师相互协作对整体的系统体系结构进行设计和演化。

3. 发布同步。对每个团队的产品发布日期进行同步，从而产生一个可演示和完整的系统。

4. Scrum 团队的每日站立会议。Scrum 团队之间每天进行站立式会议，来自各个团队的代表会讨论进度、发现问题并且计划当前要做的工作。各个团队的每日站立会议可以在时间上错开，从而使来自其他团队的代表在必要的时候可以参加。

3.4.4 面向整个组织的敏捷方法

开发软件产品的小软件公司通常是敏捷方法最热情的采用者。这些公司不会受到组织的官僚主义或过程标准的限制，并且可以为了采用新想法而快速变化。当然，更大的公司也在一些项目中尝试过敏捷方法，但是对他们而言，在整个组织范围内“扩展”这些方法的使用范围要困难得多。

向大公司引入敏捷方法之所以困难是因为以下这些原因。

1. 没有敏捷方法经验的项目经理可能担心接受新方法而带来的风险，因为他们不知道这些方法会对他们的一些特定的项目造成什么样的影响。

2. 大型组织经常都制定了所有项目都要遵循的质量规程和标准，因为他们的官僚主义状况，这些很有可能会与敏捷方法不相适应。有时候这些规程和标准由软件工具（例如，需求管理工具）来支持，并明确要求所有项目都使用这些工具。

3. 敏捷方法似乎在团队成员具有相对较高的技能水平时效果最好。然而，在大型组织中，很有可能人员的技术和能力水平参差不齐，技能水平较差的人在敏捷过程中可能很难成为一个有效的团队成员。

4. 敏捷方法可能会存在文化上的抵制，特别是在那些长期使用传统系统工程过程的组织中。

变更管理和测试规程是公司规程的两个例子，它们可能与敏捷方法不相适应。变更管理是对一个系统的变更进行控制的过程，其目的是使变更的影响可预测而成本能够得到控制。所有的变更都必须在实施之前提前得到批准，这与重构的思想相冲突。当重构成为敏捷过程的一部分时，任何开发者都可以在不需要外部批准的情况下改进任何代码。对于大型系统，

还存在测试标准，其中系统的构建版本会交给外部的测试团队。这可能与敏捷开发方法中所使用的测试先行的方法相冲突。

在一个大型组织中引入并持续使用敏捷方法是一个文化改变的过程。文化改变需要很长时间来实现，而且在实现之前经常需要管理上的变化。希望使用敏捷方法的企业需要敏捷方法的传道者来推动变化，但并不是要向不情愿的开发者强制推行敏捷方法。很多企业都发现引入敏捷的最好方法是一点点来，从一个充满热情的开发小组开始。一个成功的敏捷项目可以作为一个起点，然后项目团队再将敏捷实践扩展到整个组织。一旦敏捷的思想得到了广泛理解，那么就可以采取明确的动作将其扩展到整个组织。

要点

- 敏捷方法是迭代化开发方法，关注减少过程开销和文档化，强调增量式的软件交付。敏捷方法直接将客户代表包含在开发过程中。
- 关于使用敏捷还是计划驱动的方法进行软件开发的决定应当取决于所开发的软件类型、开发团队的能力、开发系统的企业的文化。在实践中，可以将敏捷和基于计划的技术混合使用。
- 敏捷开发实践包括将需求表达为用户故事、结对编程、重构、持续集成和测试先行的开发。
- Scrum 是一种敏捷方法，为敏捷项目的组织提供了一个框架。Scrum 开发围绕着一组冲刺开展，每次冲刺都是一个开发系统增量的固定时长的期间。制定计划的方式是对待办事项进行优先级排序并为下一个冲刺选取优先级最高的任务。
- 为了使敏捷方法规模化，必须将一些基于计划的实践与敏捷实践相集成。这些实践包括预先考虑需求、多个客户代表、更多的文档、整个项目团队采用同样的工具、在团队之间进行发布版本的同步。

阅读推荐

《Get Ready for Agile Methods, With Care》是一个有着深厚经验的软件工程师所写，其中对敏捷方法进行了深刻的评论，讨论了敏捷方法的优点和缺点。虽然已经是 15 年前的文章了，但仍然非常值得关注。(B. Boehm, IEEE Computer, January 2002) <http://dx.doi.org/10.1109/2.976920>

《Extreme Programming Explained》是第一本关于 XP 方法的书，也许至今仍然是可读性最高的一本。书中从 XP 方法的一位发明者的观点出发对该方法进行了解释，整本书都透露着他的热情。(K. Beck and C. Andres, Addison-Wesley, 2004)

《Essential Scrum: A Practical Guide to the Most Popular Agile Process》对于 Scrum 方法的发展进行了全面和易于理解的介绍。(K. S. Rubin, Addison-Wesley, 2013)

《Agility at Scale: Economic Governance, Measured Improvement and Disciplined Delivery》这篇论文介绍了 IBM 的敏捷规模化方法，其中包括一个系统性的方法来集成基于计划的开发以及敏捷开发。其中对于敏捷规模化中的关键问题进行了出色和深入的探讨。(A. W. Brown, S. W. Ambler, and W. Royce, Proc. 35th Int. Conf. on Software Engineering, 2013) <http://dx.doi.org/10.1145/12944.12948>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap3/>

支持视频的链接: <http://software-engineering-book.com/videos/agile-methods/>

练习

- 3.1 为什么新系统的快速交付和部署对于业务而言经常比这些系统的详细功能更重要?
- 3.2 敏捷方法所基于的原则是如何加快软件的开发和部署的?
- 3.3 极限编程将用户需求表达为故事, 每个故事写在卡片上。讨论这种方法对于需求描述的优点和缺点。
- 3.4 为什么测试先行的开发可以帮助程序员更好地理解系统需求? 测试先行的开发有什么潜在的困难?
- 3.5 针对结对工作的程序员的生产率为什么可能比两个单独工作的程序员的生产率的一半要多提出 4 点原因。
- 3.6 比较 Scrum 方法和传统的基于计划的方法中的项目管理(如第 23 章所介绍的)。你的比较应该基于每种方法对项目人员分配计划、项目成本估算、维持团队延续性、管理项目团队成员变化等方面的有效性。
- 3.7 为了降低计算的成本以及对于环境的影响, 你的公司决定关闭一些办公室, 并且为员工在家工作提供支持。然而, 引入该政策的高级管理层不知道团队正在使用 Scrum 方法开发软件。解释如何使用技术来支持分布式环境下的 Scrum 方法从而使在家工作成为可能。使用这一方法最有可能遇到什么样的问题?
- 3.8 为什么在将敏捷方法规模化应用到由分布式开发团队开发的更大项目中的时候, 有必要从基于计划的方法中引入一些方法和文档?
- 3.9 为什么在大型项目中使用敏捷方法开发一个将作为组织的系统之系统的一部分的新信息系统时可能很困难?
- 3.10 让一个用户紧密参与软件开发团队的一个问题是他们会被“同化”。也就是说, 他们会采用开发团队的观点, 并丢掉关于用户需要的观点。提出 3 种有利于避免这一问题的方法, 并讨论每种方法的优点和缺点。

参考文献

Ambler, S. W. 2010. "Scaling Agile: A Executive Guide." http://www.ibm.com/developerworks/community/blogs/ambler/entry/scaling_agile_an_executive_guide10/

Arisholm, E., H. Gallis, T. Dyba, and D. I. K. Sjöberg. 2007. "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise." *IEEE Trans. on Software Eng.* 33 (2): 65–86. doi:10.1109/TSE.2007.17.

Beck, K. 1998. "Chrysler Goes to 'Extremes.'" *Distributed Computing* (10): 24–28.

———. 1999. "Embracing Change with Extreme Programming." *IEEE Computer* 32 (10): 70–78. doi:10.1109/2.796139.

Bellouiti, S. 2009. "How Scrum Helped Our A-Team." <http://www.scrumalliance.org/community/articles/2009/2009-june/how-scrum-helped-our-team>

- Bird, J. 2011. "You Can't Be Agile in Maintenance." <http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html>
- Deemer, P. 2011. "The Distributed Scrum Primer." <http://www.goodagile.com/distributedscrumprimer/>.
- Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.
- Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.
- Jeffries, R., and G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.
- Kilner, S. 2012. "Can Agile Methods Work for Software Maintenance." <http://www.vlegaci.com/can-agile-methods-work-for-software-maintenance-part-1/>
- Larman, C., and V. R. Basili. 2003. "Iterative and Incremental Development: A Brief History." *IEEE Computer* 36 (6): 47–56. doi:10.1109/MC.2003.1204375.
- Leffingwell, D. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.
- Leffingwell, D. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*. Boston: Addison-Wesley.
- Mulder, M., and M. van Vliet. 2008. "Case Study: Distributed Scrum Project for Dutch Railways." *InfoQ*. <http://www.infoq.com/articles/dutch-railway-scrum>
- Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.
- Schatz, B., and I. Abdelshafi. 2005. "Primavera Gets Agile: A Successful Transition to Agile Development." *IEEE Software* 22 (3): 36–42. doi:10.1109/MS.2005.74.
- Schwaber, K., and M. Beedle. 2001. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice-Hall.
- Stapleton, J. 2003. *DSDM: Business Focused Development, 2nd ed.* Harlow, UK: Pearson Education.
- Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action, 2/e*. Greenwich, CT: Manning Publications.
- Weinberg, G. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand.
- Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. 2000. "Strengthening the Case for Pair Programming." *IEEE Software* 17 (4): 19–25. doi:10.1109/52.854064.

需求工程

目标

本章的目标是介绍软件需求，解释在发现并文档化这些需求的过程中所涉及的过程。阅读完本章后，你将：

- 理解用户和系统需求的概念，以及为什么这些需求应当以不同的方式进行描述；
- 理解功能和功能性软件需求的区别；
- 理解主要的需求工程活动，包括抽取、分析、确认，以及这些活动之间的关系；
- 理解需求管理的必要性，以及需求管理如何支持其他需求工程活动。

对一个系统的需求是关于该系统应当提供的服务以及对其运行的约束的描述。这些需求反映了客户对一个系统的需要，这种需要服务于一定的目的，例如控制一个设备、下订单或者找到所要的信息。找出、分析、文档化并且检查这些服务和约束的过程被称为需求工程 (Requirement Engineering, RE)。

术语需求 (requirement) 在软件产业中的使用并不完全一致。有些时候，一个需求只是一个关于系统应当提供的服务或者一个对于系统的约束的高层抽象陈述。而在另一个极端上，它又指的是关于系统功能的详细和正式的定义。Davis (Davis 1993) 如下解释这一区别存在的原因：

如果一个公司想通过一个合同完成一个大型的软件开发项目，那么它必须以一种足够抽象的方式定义它的要求，以避免事先定义相应的解决方案。这些需求必须写下来以使得多个承包商可以对该合同进行投标、报价，也许还可以提出不同的方式来满足客户组织的要求。一旦确定合同投标结果，承包商必须更加详细地为客户编写一个系统定义以使客户理解软件可以做什么并对其进行确认。这些文档都可以称为系统的需求文档^①。

需求工程过程中出现的一些问题是由于没有清晰地区分这些不同描述层次上的需求而造成的。为了加以区分，本书用术语用户需求来指高层的抽象需求，而用系统需求来指关于系统应当做什么的详细描述。用户需求和系统需求可以按照下面这样定义。

1. 用户需求使用自然语言和图形，陈述系统被期望向系统用户提供什么服务以及系统运行必须满足的约束。用户需求可以是对系统特征的大概陈述，也可以是关于系统功能的详细和精确的描述。

2. 系统需求是对软件系统的功能、服务和运行约束的更详细的描述。系统需求文档 (有时候被称为功能规格说明) 应该精确定义要实现哪些东西。它可以是系统购买方和软件开发者之间合同的一部分。

向不同类型的读者传达关于一个系统的信息需要不同类型的需求。图 4-1 描述了用户需求和系统需求之间的区分。这个例子出自心理健康病人信息系统 (Mentcare 系统)，其中反映了一个用户需求可以如何扩展为多个系统需求。可以从图 4-1 中看到用户需求非常笼统。系统需求提供了关于要实现的系统服务和功能的更加明确的信息。

① Davis, A. M. 1993. Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice-Hall.

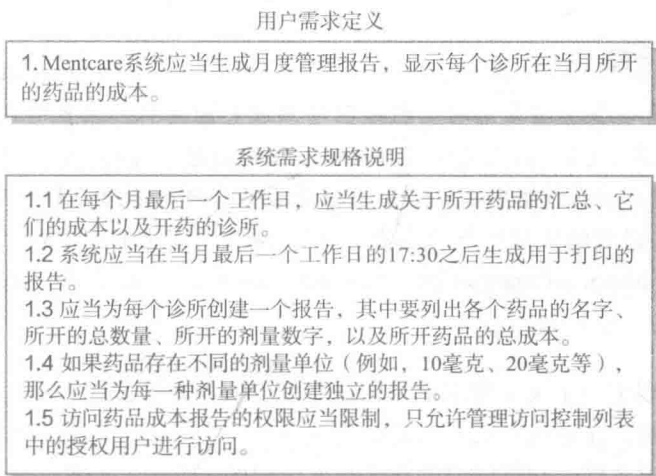


图 4-1 用户需求与系统需求

开发人员要在不同的抽象层次上书写需求，因为不同类型的读者会以不同的方式使用这些需求。图 4-2 描述了用户需求和系统需求的不同读者类型。用户需求的读者通常并不关心系统如何实现，可能是对系统的详细实现不感兴趣的管理人员。系统需求的读者需要更精确地了解系统要做什么，因为他们关心系统将如何支持业务过程或者他们本身参与系统实现。

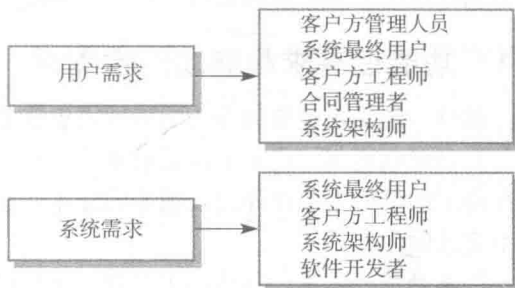


图 4-2 用户需求和系统需求所面对的不同读者类型

图 4-2 中所示的不同类型的文档读者都是系统利益相关者的例子。除了用户，还有很多其他人对于系统存在某种利益。系统利益相关者包括以某种方式受到系统影响的任何人以及在系统中存在某种合法利益的任何人。从系统的最终用户、管理人员直到对系统的可接受性进行认证的外部监管者等外部利益相关者都可能是利益相关者。例如，Mentcare 系统包括以下利益相关者。

1. 信息被记录在系统中的病人以及他们的家属；
2. 负责对病人进行评估和治疗的医生；
3. 协调医生的问诊并对某些治疗进行管理和指导的护士；
4. 管理病人预约的医疗接待人员；
5. 负责安装和维护系统的 IT 人员；
6. 必须确保系统满足当前对病人治疗的伦理指南的医疗伦理管理人员；
7. 从系统中获取管理信息的医疗管理人员；
8. 负责系统信息的维护和保存并确保记录保存规程得到适当遵循的医疗记录人员。

需求工程通常被认为是软件工程过程的第一个阶段。然而，对系统需求的一些理解可能必须在一个系统的采购或开发决策做出之前就开发出来。这一早期的需求工程建立了一个关于系统要做什么以及系统可以提供的好处的高层视图。这些需求接下来可以在可行性研究（试图从技术和经济上评价该系统的开发是否可行）中进行考虑。可行性研究的结果帮助管理层决定是否继续进行该系统的采购或开发。



可行性研究

可行性研究是一个应该在需求工程过程早期进行的简短、聚焦的研究。这个研究应该回答3个关键问题：（1）系统是否可以服务于组织的总体目标？（2）系统是否可以在进度和预算范围内用当前的技术实现？（3）系统是否可以与所使用的其他系统相集成？

如果对于这些问题的任何一个回答是“否”，那么很可能不应该继续这个项目。

<http://software-engineering-book.com/web/feasibility-study/>

在本章中，呈现了一个关于需求的“传统”观点而不是敏捷过程中的需求（见第3章）。对于大多数大型系统，在系统实现开始之前仍然存在一个清晰、可识别的需求工程阶段。该阶段的产出物是需求文档，它可以成为系统开发合同的一部分。当然，随后需求可能发生变化，用户需求会被展开为更详细的系统需求。有时，在系统开发过程中同时抽取需求的敏捷方法也可以用于细化和精化用户需求。

4.1 功能性需求和非功能性需求

软件系统需求经常被分为功能性需求或非功能性需求。

1. 功能性需求。这些需求是对系统应该提供的服务、系统应该如何响应特定的输入、系统在特定的情形中应该如何表现等的陈述。在某些情况下，功能性需求还可以明确地陈述系统不应该做什么。

2. 非功能性需求。这些需求是对系统提供的服务或功能的约束，包括时间性约束、对于开发过程的约束、标准规范中所施加的约束等。非功能性需求经常适用于系统整体而不是单个的系统特征或服务。

在现实中，不同类型的需求之间的区别并不像这里的简单定义所表达的那样清楚。一个关注信息安全的用户需求，例如，一个关于授权用户访问权限，可能看起来像是一个非功能性需求。然而，当开发得更加详细之后，这个需求可能会产生其他一些明显是功能性需求的需求，例如，要求在系统中包含用户认证的功能。

这表明需求不是独立的，一个需求经常会产生其他需求或对其他需求产生约束。因此，系统需求并不只是刻画所需要的系统服务或特征；它们也要刻画确保这些服务/特征能够有效交付的必要功能。



领域需求

领域需求是从系统的应用领域而不是从系统用户的特定需要中得出的。它们可以自身就是新的功能性需求、对于已有的功能性需求的约束，或者陈述特定的计算必须如何进行。

领域需求的问题是软件工程师可能不理解系统运行所属的领域的特性。这意味着这些工程师可能不知道一个领域需求是否漏掉了或者与其他需求相冲突。

<http://software-engineering-book.com/web/domain-requirements/>

4.1.1 功能性需求

系统的功能性需求描述系统应该做什么。这些需求取决于所开发的软件的类型、软件所期望的用户，以及该组织在书写需求时通常所采取的方法。当被表达为用户需求时，功能性需求应当用自然语言描述，以使得系统用户和管理人员能够理解它们。功能性系统需求将用户需求展开，是面向系统开发者描述的，应该详细描述系统功能，系统的输入、输出和异常。

功能性系统需求可以是关于系统应该做什么的大体上的需求，也可以是反映局部的工作方式或者组织已有系统的非常特定的需求。例如，下面是 Mentcare 系统功能性需求的例子，用于维护那些接受心理健康问题治疗的病人的信息。

1. 用户应当能够搜索到所有诊所的预约列表；
2. 系统应当每天为每个诊所生成一个希望预约当天看诊的病人列表；
3. 应当通过 8 位数字的雇员编号对使用该系统的每个工作人员进行唯一标识。

这些用户需求定义了系统中应当包含的一些特定的功能。这些需求表明功能性需求可以在不同的抽象层次上进行描述（对比需求 1 和 3）。

功能性需求，顾名思义，传统意义上关注系统应该做什么。然而，如果一个组织确定一个已有的成品系统软件产品可以满足其需要，那么开发一个详细的功能性规格说明的意义不大。在这种情况下，关注点应该集中在信息需求的开发，即刻画人们完成自己的工作所需的信息。信息需求刻画了所需的信息以及信息是如何提供和组织的。因此，一个对于 Mentcare 系统的信息需求可以刻画为希望预约当天看诊的病人列表中要包含哪些信息。

需求规格说明中的不精确可能导致客户和软件开发者之间的争执。对于系统开发者而言，一种很自然的情况是按照简化实现的方式去解读一个模糊的需求。然而，这经常不是客户所要的。为此必须建立新的需求并对系统进行修改。显然，这会拖延系统的交付并增加成本。

例如，上面的列表中的第一条 Mentcare 系统需求说用户应当能够搜索所有诊所的预约列表。这条需求背后的考虑是有心理健康问题的病人有时候思维混乱，他们可能会在某一个诊所预约但实际上去了另外一家诊所。如果他们预约，那么他们将会被记录为已预约而不管是哪个诊所。

提出搜索需求的医护工作人员所期望的“搜索”可能是给定一个病人名字，系统在所有诊所的所有预约中查找这个名字。然而，这一点在需求描述中并不明确。系统开发者可能会按照最容易实现的方式理解这条需求。他们所理解的搜索功能可能需要用户选择一个诊所然后搜索预约了这个诊所的病人。这需要更多的用户输入，因此需要花更长的时间来完成搜索。

理想情况下，一个系统的功能性需求规约应当是完整、一致的。完整性意味着用户所需要的所有的服务和信息都应该被定义；一致性意味着需求不应当自相矛盾。

在实践中，只有可能对非常小的软件系统实现需求的一致性和完整性。其中的一个原因是，在给大型、复杂系统编写规格说明时很容易犯错误和遗漏东西。另一个原因是大型系统有很多利益相关者，他们的背景和期望各不相同。这些利益相关者很有可能会有不同的并且通常是不一致的需求。这些不一致的需求在最初编写需求规格说明时可能不明显，只有在经过更深入的分析后或者在系统开发过程中才会被发现。

4.1.2 非功能性需求

顾名思义,非功能性需求是指与系统向其用户提供的特定服务不直接相关的需求。这些非功能性需求通常会刻画或约束系统的整体特性。它们可能会与系统的涌现特性(例如可靠性、响应时间、存储使用)相关。或者,它们也可以定义对系统实现的约束,例如,输入/输出设备的能力,或者与其他系统的接口中所使用的数据表示。

非功能性需求经常比单个的功能性需求更关键。对于一个实际上不符合他们需要的系统功能,系统用户通常都可以找到变通的办法来实现自己的目的。然而,无法满足一个非功能性需求可能意味着整个系统都没法用。例如,如果一个飞机系统不满足可靠性需求,那么该系统不会通过安全运行认证;如果一个嵌入式控制系统无法满足其性能需求,那么控制功能将无法正确运行。

哪个系统构件实现了特定的功能性需求(例如,可能会有格式化构件来实现报表需求)经常是可以确定的。但是,确定哪些系统构件实现了非功能性需求通常更加困难。这些非功能性需求的实现经常跨越整个系统,这主要是因为以下两点原因。

1. 非功能性需求可能会影响一个系统的整个体系结构而非单个的构件。例如,为了确保一个嵌入式系统满足性能需求,体系结构设计可能要考虑通过合理的系统结构来尽可能减少构件间的通信。

2. 一个非功能性需求,例如信息安全需求,可能会产生多个相互联系的功能性需求,这些功能性需求定义了实现该非功能性需求所需要的新的系统服务。此外,非功能性需求还有可能产生对已有需求构成约束的新需求,例如,限制对系统中信息的访问。

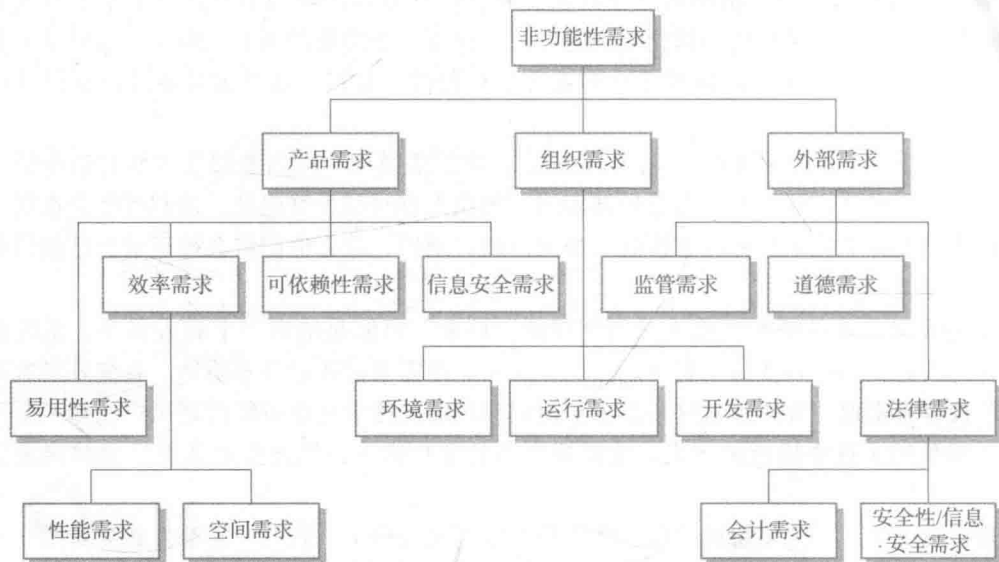


图 4-3 非功能性需求的类型

非功能性需求的产生源自于由预算约束导致的用户要求、组织政策、与其他软件或硬件系统的互操作要求,或者安全性监管或隐私法规等外部因素。图 4-3 对非功能性需求进行了分类。从图中可以看出非功能性需求可能来自于所要求的软件特性(产品需求)、开发软件的组织(组织需求),或者一些外部来源,具体如下。

1. 产品需求。这些需求刻画或者约束了软件的运行时行为。例如，效率需求可以要求系统必须运行得多快以及系统需要多少存储；可依赖性需求可以设定可接受的系统失效率；此外还有信息安全需求以及可用性需求。

2. 组织需求。这些需求是源自于客户和开发方组织的政策和规程的一类宽泛的系统需求。例如，运行需求可以定义系统将如何被使用；开发需求可以要求所使用的编程语言、开发环境或者过程标准；环境需求可以刻画系统的运行环境。

3. 外部需求。这一宽泛的类型覆盖了源自于系统及其开发过程之外的因素的所有需求。可以包括由监管者（例如，核安全管理机构）设定的系统通过使用许可必须要做的事情的监管需求；必须遵守以确保系统运行符合法律的法律需求；确保系统可以被用户以及公众接受的道德需求。

图 4-4 描述了可以包含在 Mentcare 系统需求规格说明中的产品、组织和外部需求的例子。其中的产品需求是一个定义了系统必须开放的时间以及所允许的每天宕机时间的可用性需求。其中没有提到任何关于 Mentcare 系统的功能的内容，而是清楚地明确了一个系统设计者必须考虑的约束。

产品需求

Mentcare 系统应当在常规工作时间（周一至周五，08:30-17:30）中对所有诊所都是可用的。任何一天在常规工作时间之内的宕机时间不应该超过 5 秒。

组织需求

Mentcare 系统用户应当使用他们的健康管理机构身份卡进行身份认证。

外部需求

系统应当按照 HStan-03-2006-priv 中的要求实现病人隐私条款。

图 4-4 Mentcare 系统非功能性需求的例子

组织需求刻画了用户如何向系统验证自己的身份。运营该系统的健康管理机构正在转向让所有的软件都执行标准的身份认证规程，以替代原有的用户名登录的方法。用户通过在读卡器上刷自己的身份卡来进行身份认证。外部需求源自于系统符合隐私法律的要求。隐私在医疗健康系统中显然是一个非常重要的问题，该需求要求系统开发应当符合国家的隐私标准。

与非功能性需求相关的一个共性问题是，利益相关者将这些需求表述为泛化的目标，例如，容易使用、系统从失效中恢复的能力、快速用户响应等。目标表达了一些好的意图，但是对系统开发者而言会导致一些问题，因为会存在模糊的理解空间，并且会在系统交付时导致争议。例如，下面这个系统目标是管理人员表达可用性需求的常见方式。

系统应当对医护人员而言容易使用，并且应当能通过某种方式进行组织，以使得用户错误能够最小化。

这个例子可以用于表明目标可以表达为一个“可测试的”非功能性需求。这个系统目标无法客观地验证，但是根据下面的描述至少可以在系统测试时使用软件插装来对用户所犯的错误进行计数。

医护人员应当在 2 小时的培训后有能力使用所有的系统功能。接受培训后，有经验的用户所犯的错误的平均数量不应当超过每小时（系统使用时间）2 个。

只要有可能，你都应该定量地描述非功能性需求以使得这些需求可以客观地测试。

图 4-5 描述了可以用来刻画非功能性系统属性的度量指标。可以在系统测试时对这些特性进行度量以检查系统是否满足非功能性需求。

属 性	度量指标
速度	每秒处理的事务 用户 / 事件响应时间 屏幕刷新时间
规模	兆字节 / 只读存储器芯片数量
易于使用	培训时间 帮助画面的数量
可靠性	平均失效时间 不可用的概率 失效发生率 可用性
鲁棒性	失效后重启时间 导致失效的事件百分比 失效时数据损坏的概率
可移植性	依赖于目标的陈述百分比 目标系统的数量

图 4-5 刻画非功能性需求的度量

在实践中，系统的客户经常感觉很难将自己的目标表达为可度量的需求。对于一些目标，例如可维护性，并不存在可用的简单的度量指标。在其他情况下，即使有可能得到量化的规格说明，客户可能也无法将自己的需要与这些规格说明联系起来。例如，他们根据自己对计算机系统的日常经验无法理解一些定义了可靠性的数字是什么意思。而且，客观地验证可度量的非功能性需求的代价可能会很高，为系统付款的客户可能并不认为这些代价是合理的。

非功能性需求经常与其他的功能性或非功能性需求冲突或交互。例如，图 4-4 中的身份认证需求要求连接系统的每一台计算机都要安装一个读卡器。然而，可能存在另一条需求要求支持从医生或护士的平板电脑或智能手机上远程访问系统。这些移动设备上一般都没有读卡器，因此在这种情况下可能要支持其他身份认证方法。

在需求文档中很难将功能性和非功能性需求分开。如果非功能性需求与功能性需求是分开描述的，那么它们之间的关系可能很难理解。然而，理想情况下应该突出那些明显与涌现性系统属性相关的需求，例如，性能或可靠性。可以将它们放在需求文档中的一个独立的章节中，或者通过某种方式将它们与其他系统需求相区分。

非功能性需求，如可靠性、安全性和机密性需求，对于关键性系统尤为重要。这些可依赖性需求将在第二部分中讨论，其中将介绍刻画可靠性、安全性和信息安全需求的方式。

4.2 需求工程过程

如第 2 章所述，需求工程包括 3 个关键活动：通过与利益相关者交互发现需求（抽取和分析）；将这些需求转换为标准格式（规格说明）；检查需求是否实际上定义了客户所要的系统（确认）。图 2-4 中以顺序的方式描述了这些活动。然而，在实践中需求工程是一个迭代化的过程，其中的活动相互交织。

图 4-6 描述了这种相互交织。这些活动被组织为围绕着一个螺旋的迭代过程。需求工程过程的输出是一个系统需求文档。一次迭代中所花费的时间和工作量取决于整个过程的阶段、所开发的系统类型以及可用的预算。

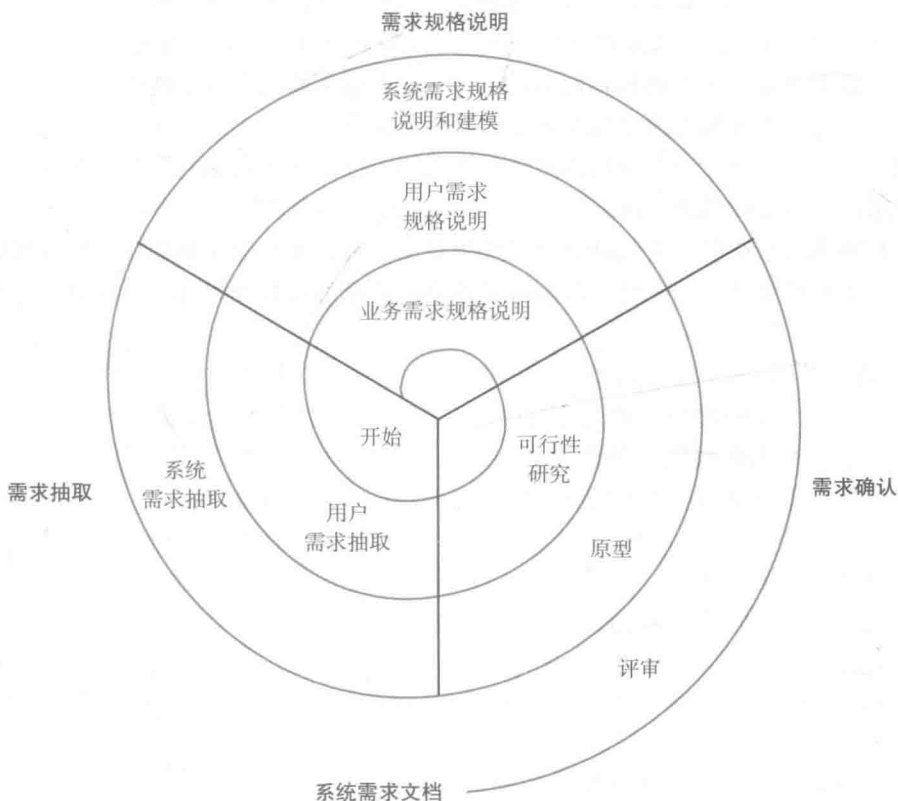


图 4-6 需求工程过程的螺旋视图

在过程的早期阶段，大多数工作量花在理解高级业务和非功能性需求以及系统的用户需求上。此后，在螺旋靠外的环上更多的工作量将用于抽取和理解非功能性需求以及更详细的系统需求。

在这一螺旋模型所支持的各种开发方法中，需求可以被开发到不同的抽象层次。该螺旋上的迭代次数可以各不相同，从而在一些或所有的用户需求都被抽取后，可以从螺旋中退出。敏捷开发可以代替原型来使用，使需求和系统实现可以一起开发。

事实上所有的系统中的需求都会变化。系统中所涉及的人会得出关于他们想让软件做什么的更好的理解；购买系统的组织发生变化；系统的硬件、软件和组织环境发生改变。必须对变更进行管理以理解它们对其他需求的影响，以及实施变更的成本和对系统的影响。4.6 节将介绍需求管理过程。

4.3 需求抽取

需求抽取过程的目的是，理解利益相关者所做的事情以及他们会如何使用一个新系统来支持他们的工作。在需求抽取过程中，软件工程师与利益相关者一起工作来搞清楚应用领域、工作活动、利益相关者想要的服务和系统特征、系统要达到的性能、硬件约束等。

从系统利益相关者那里抽取和理解需求是一个困难的过程，原因如下。

1. 利益相关者经常不知道他们想从一个计算机系统中得到什么, 除了一些非常泛泛的说法; 他们可能会觉得很难表达他们想让系统做的事情; 他们可能会提出一些不切实际的要求, 因为他们不知道哪些可行哪些不可行。

2. 一个系统中的利益相关者会很自然地用他们自己的话来表达需求, 其中隐含着一些关于他们自己工作的知识。需求工程师对于客户的业务领域没有经验, 可能无法理解这些需求。

3. 不同的利益相关者有各种不同的需求, 他们会以不同的方式表达他们的需求。需求工程师必须发现所有潜在的需求来源并且发现共性和冲突。

4. 政治性因素可能影响系统的需求。管理人员可能会出于增加自己在组织中的影响力的原因而要求某些特定的系统需求。

5. 进行需求分析时所处的经济和业务环境是动态的, 不可避免地会在分析过程中发生变化。特定需求的重要性可能变化。新的需求可能会从此前没有咨询过的新利益相关者那里涌现出来。

一个抽取和分析过程的过程模型如图 4-7 所示。每个组织都可以在这个通用模型基础上, 基于人员经验、所开发的系统类型、所使用的标准等本地因素定义自己的过程。

其中的过程活动如下。

1. 需求发现和理解。在此过程中与系统利益相关者进行交互以发现他们的需求。来自利益相关者和文档的领域需求也在此活动中发现。

2. 需求分类和组织。在此过程中处理所收集的未整理需求, 将相关的需求进行分组并将它们组织为内聚的聚类。

3. 需求优先级排序和协商。当涉及多个方面的利益相关者时会不可避免地发生需求冲突。该活动关注对需求进行优先级排序, 找出并通过协商解决需求冲突。通常, 利益相关者必须一起协商以解决分歧并做出妥协、达成一致。

4. 需求文档化。对需求进行文档化并提供给下一轮螺旋。该阶段可以产生一个软件需求文档的早期草稿, 或者直接通过白板、Wiki 或者其他共享空间等非正式的方式保存需求。

图 4-7 显示需求抽取和分析是一个迭代化过程, 包含从一个活动到其他活动的持续反馈。过程循环开始于需求发现, 结束于需求文档化。分析人员对需求的理解经过每次循环后都会得到改进。当需求文档产生后, 该循环结束。

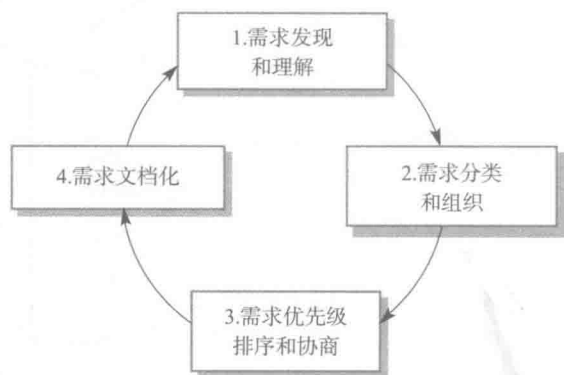


图 4-7 需求抽取和分析过程



视角

视角是一种收集和组织需求的方式, 这些需求来自具有某些共性的一组利益相关者。因此, 每个视角包括一组系统需求。视角可能来自于最终用户、管理人员或其他人, 帮助识别可以提供需求信息的人以及组织需求用于分析。

<http://www.software-engineering-book.com/web/viewpoints/>

为了简化需求的分析,对利益相关者的信息进行组织和分组是很有帮助的。其中一种方式是,将每个利益相关者组考虑为一种视角,并且将从该组中收集的所有需求作为该视角的内容。还可以用视角来表示领域需求以及来自其他系统的约束。此外,也可以使用一个系统体系结构模型来识别子系统,并将需求与每个子系统相关联。

不可避免地,不同的利益相关者对需求的重要性和优先级有不同的观点,有时候这些观点会相互冲突。如果一些利益相关者感到他们的观点没有得到适当的考虑,那么他们可能试图去故意阻碍需求工程过程。因此,组织日常的利益相关者会议是很重要的。利益相关者应该有机会表达自己的关注点并就需求折中取得一致意见。

在需求文档化阶段,使用简单的语言和图形来描述需求是很重要的。这使得利益相关者可以理解这些需求并发表评论。为了让信息共享更容易,最好使用所有感兴趣的利益相关者都能访问的共享文档(例如使用 Google Docs 或者 Office 365)或者 Wiki。

4.3.1 需求抽取技术

需求抽取包含与不同类型的利益相关者交谈以发现关于待开发系统的信息。可以补充关于现有系统及其使用的知识,以及来自不同种类文档的信息。你需要花时间理解人们如何工作、他们生产什么、他们如何使用其他系统,以及他们要如何变化以适应新系统。

需求抽取有两个基本的方法。

1. 访谈,开发者和其他人谈论他们做的事情。
2. 观察或人种学调查,观察人们做自己的工作来了解他们使用哪些制品、他们如何使用这些制品等。

应当将访谈和观察相结合来收集信息,并且从中得出需求,这些信息接下来将成为进一步讨论的基础。

访谈

大多数需求工程过程都包括与系统利益相关者的正式或非正式的访谈。在这些访谈中,需求工程团队针对利益相关者当前使用的系统以及将开发的系统提出问题。从这些问题的回答中得出需求。访谈可以有两种类型。

1. 封闭式访谈,利益相关者回答一组预定义的问题。
2. 开放式访谈,没有预定义的日程,需求工程团队与系统利益相关者探索一系列问题,并得到对他们需要的更好的理解。

在实践中,与利益相关者的访谈通常都会结合这两种方式。最初,可能会针对一些特定的问题寻求答案,但这些问题通常会引发其他一些问题,这些问题会以更加非正式的方式进行探讨。完全开放式的讨论通常效果都不好。常常要通过问一些问题来开始整个过程,并且让访谈聚焦于待开发的系统。

访谈对于获得一个总体理解是有好处的,包括:利益相关者做什么,他们会如何与新系统交互,他们使用当前系统时遇到的困难。人们喜欢谈论他们自己的工作,因此通常都会很高兴参加访谈。然而,除非有一个系统原型来进行演示,否则不应该期望利益相关者说出太多特定的和细节的需求。每个人都觉得很难形象化地描述系统应该是什么样的。需求工程团队要分析所收集的信息并从中产生需求。

通过访谈抽取领域知识可能很难,这是由于以下原因。

1. 所有的应用专家都使用自己工作领域中的专业术语。对于他们而言,在不使用这些术

语的情况下讨论领域需求是不可能的。他们通常会以一种精确且微妙的方式来表达，这可能会使需求工程师产生误解。

2. 一些领域知识对于利益相关者而言非常熟悉，以至于他们要么觉得很难解释要么认为过于基础、不值一提。例如，对一个图书馆管理员来说，所有采购的图书在加入图书馆之前要进行分类是不需要提及的。然而，这对于访谈人却不是那么显而易见，因此在需求中就没有考虑。

访谈不是一个有效的抽取组织需求及约束的技术，因为组织中不同的人之间存在着微妙的权力关系。公开的组织结构很少与组织中实际的决策权力结构相匹配，但是受访人可能不愿意向一个陌生人揭露实际结构而不是理论上的结构。总而言之，大部分人通常不愿意谈论政治和组织性话题，这些可能会影响需求。

为了成为一个卓有成效的访谈人，应当记住以下两点。

1. 应该虚心倾听，避免预设一些关于需求的想法，并且愿意倾听利益相关者的意见。如果利益相关者提出一些意想之外的需求，那么你应该愿意改变自己关于系统的想法。

2. 应该通过使用跳板性的问题、需求建议或者一起来讨论一个原型系统等方法来提示受访人，使得讨论可以进行。对别人说“告诉我你想要什么”很难得到有用的信息。他们会觉得在一个定义好的上下文语境中谈论问题比在一个泛化的上下文语境中要容易得多。

来自访谈的信息应该与来自描述业务过程或已有系统的文档、用户观察、开发者经验等关于系统的其他信息一起使用。有时候，除了系统文档中的信息，访谈信息可能是关于系统需求的唯一的信息来源。然而，访谈本身有可能会丢失一些基本的信息，因此访谈应当与其他需求抽取技术一起使用。

人种学调查

软件系统并不是独立存在的，而是在一个社会化和组织环境中使用的，因此，软件系统需求可以由该环境产生或受其约束。很多软件系统交付后从来未被使用的一个原因是，收集的需求没有适当考虑社会和组织因素如何影响系统的实际运行。因此，在需求工程过程中很重要的一点是，要尽量去理解影响系统使用的社会和组织问题。

人种学调查是一种观察技术，可以用来理解运行过程，并且帮助得出支持这些过程的软件需求。一个分析人员让自己深入系统未来使用的工作环境，观察日常工作，记录参与者所进行的各种实际任务。人种学调查的价值是，可以帮助发现一些反映人们实际工作方式的隐含系统需求，这些需求在组织定义的正式的业务过程中并不存在。

人们经常感觉很难清晰地描述关于自己工作的详细信息，因为他们对于这些工作已经习以为常。他们理解自己的工作，但可能不理解他们的工作与组织中其他工作的关系。一些影响工作的社会和组织因素对于个人而言并不明显，这些因素只有在一个客观的观察者注意到之后才会变得比较清楚。例如，一个工作组具有良好的自组织性，因此各个成员了解相互的工作，在有人缺席的情况下可以相互代替。这一点可能不会在面谈中提及，因为这些成员可能没有将这一点作为他们工作的一个重要部分。

Suchman (Suchman 1983) 是使用人种学调查研究办公室工作方面的先驱。她发现实际的工作实践远远比办公自动化系统所假设的简单模型更丰富、更复杂也更动态化。假设的工作方式与实际工作方式的差异是这些办公系统对于生产力没有产生显著影响的最重要的原因。Crabtree (Crabtree 2003) 对这之后的很多研究进行了探讨，并大致介绍了人种学调查在系统设计中的应用。本书作者曾研究过在软件工程过程中使用一些有趣的人种学调查方

法——将人种学调查与需求工程方法联系起来 (Viller and Sommerville 2000), 并描述协作式系统中的交互模式 (Martin and Sommerville 2004)。

人种学调查对于发现以下两种类型的需求特别有效。

1. 从人们的实际工作方式 (而不是业务过程定义中所说的工作方式) 中得出的需求。在实践中, 人们很少遵循正式的过程。例如, 空中交通管制人员可能会关闭一个侦测飞行路线是否存在交叉的飞机碰撞警报系统, 即使常规的控制规程中要求使用该警报系统。原因也许是这个碰撞警报系统很敏感, 在飞机相距很远的时候都会发出很响的警报。控制人员可能会觉得这个警报会令人分心, 因此宁愿使用其他策略来确保飞机不会出现在可能发生碰撞的路线上。

2. 从与他人的合作以及对其他人的活动的了解中得出的需求。例如, 空中交通管制人员 (Air Traffic Controller, ATC) 可能会通过对其他控制人员工作的了解来预测将会进入他们的控制区域的飞机的数量。接下来他们会根据所预测的负载量来修改他们的控制策略。因此, 一个自动化的 ATC 系统应当允许一个控制区域中的控制人员对于相邻控制区域中的工作有一些了解。

人种学调查可以与系统原型的开发结合起来使用 (见图 4-8)。人种学调查为原型的开发提供信息, 从而使其所需要的原型精化循环更少。此外, 原型化通过识别接下来可以与调查人员探讨的问题和疑问来使得人种学调查更加聚焦。之后, 他应当在系统研究的下一个阶段中寻求对于这些问题的答案 (Sommerville et al. 1993)。

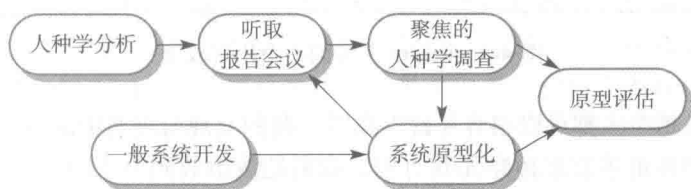


图 4-8 用于需求分析的人种学调查与原型化

人种学调查有助于理解现有系统, 但这种理解并不总是有助于创新。创新与新产品开发尤为相关。有评论提到诺基亚公司当时使用人种学调查来发现人们使用手机的方式, 并基于此开发新的手机模型; 然而另一方面, 苹果公司则忽略当前的使用情况而是革命性地通过引入 iPhone 颠覆了手机产业。

人种学调查可以揭示经常被其他需求抽取技术忽略的关键过程细节。然而, 由于关注最终用户, 该方法对于发现更广阔的组织或领域需求或提出创新就没那么有效了。因此, 人种学调查应该被用作一系列需求抽取技术中的一种。

4.3.2 故事和场景

人们发现, 与抽象描述相比, 与现实生活中的例子相联系要容易得多。他们不善于告诉你系统需求。但是, 他们也许可以描述他们如何处理特定的情形, 或者想象他们可能以一种新的工作方式做事情。故事和场景是捕捉此类信息的手段。因此, 可以在与利益相关者组访谈, 或者与其他利益相关者讨论系统以及开发更特定的系统需求时, 使用故事和场景。

从本质上看, 故事和场景是一样的东西, 它们描述了系统可以如何用于一些特定的任务。故事和场景描述了人们做什么, 他们使用和产生什么信息, 以及在此过程中他们可以使

用的系统。故事和场景的区别在于描述的组织方式以及所呈现的抽象层次。故事被描述为叙述性的文本，并且呈现一种关于系统使用的高层描述；场景通常按照所收集的特定信息（例如，输入和输出）进行组织。故事对于设定系统的“概貌”很有效；故事的一些部分可以接下来被细化并表示为场景。

图 4-9 是一个故事的例子。这个例子可以用于理解第 1 章中介绍的 iLearn 数字化学习环境系统需求。这个故事描述了一所小学中的情形，其中教师使用该环境来支持学生关于渔业的项目。可以看到这是一个高层的描述。其目的是便于讨论 iLearn 系统可以如何使用，以及作为抽取该系统需求的一个起始点。

教室中的图片共享

Jack 是苏格兰北部乡村 Ullapool 的一位小学教师。他布置了一个关注该地区渔业的课程项目：了解渔业的历史、发展和经济影响。作为该项目的一部分，学生们要收集并分享来自亲属的回忆录，使用报纸档案，收集与该地区渔业和渔业团体相关的旧照片。小学生使用 iLearn 系统的 wiki 将渔业故事收集到一起，使用 SCRAN（一个历史资源站点）来访问报纸档案和照片。然而，Jack 还需要一个照片分享站点，因为他想让小学生们相互交换照片并进行评论，并且可以上传家里旧照片的扫描件。

Jack 向一个小学教师的群组（他是成员之一）发送了一封电子邮件，询问谁能推荐一个合适的系统。两位教师答复了，他们都建议使用 KidsTakePics——一个允许教师检查并监管内容的照片分享网站。由于 KidsTakePics 没有和 iLearn 系统的身份认证服务集成，他开设了一个教师和一个班级账号。他使用 iLearn 的设置服务来将 KidsTakePics 添加到小学生在班级中可以看到的服务中，这样当学生登录时就可以立即使用系统来上传来自他们的移动设备和班级计算机中的照片。

图 4-9 iLearn 系统的一个用户故事

故事的好处是每个人都可以很容易建立联系。我们发现与现实中的访谈相比，这一方法对于从范围更广的社群中获取信息尤其有用。我们把故事放到 Wiki 上，并邀请来自全国的教师和学生上面做评论。

这些高层的故事没有进入到系统的细节，但是它们可以被进一步细化为更加特定的场景。场景是对示例性的用户交互会话的描述。我认为最好以一种结构化的方式而不是叙述性文本的方式呈现场景。极限编程等敏捷方法中使用的用户故事实际上是描述性的场景，而不是帮助抽取需求的一般性的故事。

一个场景开头是对交互的概览。在抽取过程中增加细节以创建一个完整的交互描述。一个场景通常可以包括：

1. 场景开始时对系统及用户所期望的条件描述；
2. 对场景中常规的事件流的描述；
3. 关于哪些可能出错以及如何解决问题的描述；
4. 关于可能同时进行的活动的信息；
5. 场景结束时对系统状态的描述。

作为一个场景的例子，图 4-10 描述了当一个学生上传照片到 KidsTakePics 系统时（如图 4-9 中所解释的）发生的事情。该系统与其他系统的关键差别是，教师监管所上传的照片以检查照片是否适合分享。

可以看到，这个场景描述比图 4-9 中的故事要详细得多，因此可以用于提出 iLearn 系统的需求。像故事一样，可以利用场景与可能有不同工作方式的利益相关者进行讨论。

上传照片到 KidsTakePics

最初的假设：一个用户或一组用户有一个或多个数字照片要上传到图片共享网站。这些照片保存在平板电脑或笔记本电脑上。用户已经成功登录到 KidsTakePics 上。

常规：用户选择上传照片，系统提示用户在电脑上选择要上传的照片，并且选择保存照片所属的项目名称。还应当给用户一个选项，来输入应当与每个上传的照片相关联的关键词。所上传的照片通过将用户名与本地电脑上的照片文件名相拼接来命名。

上传完成后，系统自动发送一封电子邮件到项目监管者那里，请他们检查新上传的内容，并向用户生成一条屏幕消息告知该检查已完成。

可能出现的的问题：所选择的项目没有关联的监管者。自动生成一封电子邮件给学校管理员，请他们任命一个项目监管者。应当通知用户他们的照片会延迟公开访问。

具有相同名字的照片已经被同一个用户上传。应当询问用户是否希望：重新上传同名的照片，重命名照片，或者取消上传。如果用户选择重新上传照片，原来的照片会被覆盖。如果他们选择重命名照片，系统通过向已有的文件名增加一个数字来自动生成一个新名字。

其他活动：监管者可以登录到系统中，并可以在照片上传时批准其共享。

完成时的系统状态：用户登录成功。所选择的照片已经上传并且状态是“等待批准”。照片对于监管者和上传照片的用户可见。

图 4-10 在 KidsTakePics 中上传照片的场景

4.4 需求规格说明

需求规格说明是在需求文档中撰写用户和系统需求的过程。理想情况下，用户和系统需求应当是清晰、无二义、易于理解、完整和一致的。在实践中，这几乎是不可能完全实现的。利益相关者会以不同的方式解读需求，而需求中经常会存在固有的冲突和不一致。

用户需求几乎总是用自然语言书写，然后辅以需求文档中适当的图形和表格。系统需求也可以用自然语言书写，但是也可以使用基于表格、图形或数学系统模型的表示法。图 4-11 归纳了书写系统需求的几种可能的表示法。

表示法	描 述
自然语言句子	使用数字编号的自然语言句子来书写需求。每个句子应当表达一条需求
结构化自然语言	基于一个标准的表格或模板用自然语言书写需求。每个字段提供关于需求的一个方面的信息
图形化表示法	定义系统的功能性需求，辅以文本注释的图形化模型。统一建模语言（unified modeling language, UML）用况和顺序图被广泛使用
数学规格说明	这些表示法基于有限状态机或集合等数学概念。虽然这些无二义的规格说明可以减少需求文档中的二义性，但是大多数客户不理解形式化的规格说明。他们无法检查其是否表达了他们的想法，因此不愿意将其作为系统合约接受。该方法将在第 10 章中进行讨论，这一章介绍系统可依赖性

图 4-11 书写系统需求的表示法

系统的用户需求应当描述功能性需求和非功能性需求，以使不具有详细的技术知识的系统用户也可以理解。理想情况下，这些需求应当只刻画系统的外部行为。需求文档不应该包含系统体系结构或设计的细节。因此，如果你在书写用户需求，你不应该使用软件术语、结构化表示法或者形式化表示法。你应该用自然语言书写用户需求，带有简单的表格、表单和直观的图形。

系统需求是用户需求的详述版本，软件工程师将其用作系统设计的起始点。系统需求增

加了细节并解释了系统应当如何提供用户需求。它们可以作为系统实现的合约的一部分，因此，系统需求应当是对整个系统的完整以及详细的规格说明。

理想情况下，系统需求应当只描述系统的外部行为及其运行约束。它们不应当关注系统如何设计和实现。然而，在完整刻画一个复杂软件系统所需的详细程度上，排除所有的设计信息是不现实的也不合理。理由如下。

1. 你可能要设计一个初始的系统体系结构来帮助组织需求规格说明。系统需求按照构成系统的不同子系统进行组织。我们在定义 iLearn 系统需求时做了这件事，其中所提出的体系结构如图 1-8 所示。

2. 在大多数情况下，系统必须与现有的系统互操作，这对设计构成了约束并对新系统施加了额外的需求。

3. 可能有必要使用特定的体系结构来满足非功能性需求，例如通过 N 版本编程来实现可靠性（参见第 11 章）。需要对系统的安全性进行认证的外部监管者可能会要求使用一个已认证的体系结构设计。

4.4.1 自然语言规格说明

自 20 世纪 50 年代开始自然语言就被用于书写软件需求。自然语言表达能力强、直观、具有普适性。但自然语言也具有潜在的模糊性和二义性，对于自然语言的解读取决于读者的背景。其结果是，人们已经提出了很多其他书写需求的方式。然而，这些提议没有哪一个得到了广泛采用，自然语言将继续作为最广泛使用的系统和软件需求规格说明手段。

为了在书写自然语言需求时尽量减少误解，推荐你遵循以下这些简单的指南。

1. 使用一种标准格式并确保所有的需求定义都遵循该格式。标准化格式使得遗漏不太会发生，同时需求可以更容易检查。建议在可能的情况下，应当尽量用一两句话的自然语言书写需求。

2. 以一致的方式使用语言，使得必须满足的以及期望满足的需求能够区分开。必须满足的需求是系统必须支持的需求，通常使用“必须”（shall）来表达。期望满足的需求不是必不可少的，通常使用“应该”（should）来表达。

3. 使用强调性的文本（**粗体**、*斜体*或颜色）来突出需求中的关键部分。

4. 不要假设读者理解技术性的软件工程语言。“体系结构”和“模块”这样的词语很容易被误解。只要有可能，尽量避免使用专业术语、缩写和首字母缩略词。

5. 只要有可能，都应当尽量将每个用户需求与其原理关联起来。原理应当解释为什么这项需求被包含进来以及谁提出了该需求（需求来源），这样你就可以知道在需求要发生变化时咨询谁。需求原理在需求发生变化时尤其有用，因为它可以帮助判断哪些变化是不适宜的。

图 4-12 描述了这些指南该如何使用。其中包括两个第 1 章中所介绍的自动胰岛素泵嵌入式软件的需求。该嵌入式系统的其他需求在胰岛素泵的需求文档中定义，可以从本书的网页上下载得到。

3.2 系统必须每 10 分钟测量一次血糖，如果需要的话就供应一次胰岛素。（血糖的变化相对比较慢，因此不需要更频繁的测量；测量间隔时间过长的话会导致不必要的高血糖水平。）

3.6 系统必须每分钟运行一次例行的自检，测试表 1 定义的条件，并执行表 1 中所定义的相关动作。（例行的自检可以发现硬件和软件问题并警告用户常规操作可能有问题。）

图 4-12 胰岛素泵软件系统需求示例



使用自然语言进行需求规格说明的问题

自然语言的灵活性对于规格说明很有用，但也经常导致问题。自然语言需求存在不清楚的可能性，读者（设计者）可能会误解需求，因为他们与用户的背景不同。很容易将多个需求混合到一条语句中，而组织自然语言需求可能很困难。

<http://software-engineering-book.com/web/natural-language/>

4.4.2 结构化规格说明

结构化自然语言是一种书写系统需求的方式，即使用标准的方式而不是自由文本的方式书写需求。这种方法保持了自然语言大部分的表达能力和可理解性，但同时保证了规格说明具有一定的统一性。结构化语言表示法使用模板来刻画系统需求。规格说明可以使用编程语言元素来表示可选项和迭代，可以使用底纹或不同的字体来强调关键元素。

Robertson (Robertson and Robertson 2013) 在他们关于 VOLERE 需求工程方法的书中推荐先在卡片上书写用户需求，每张卡片一条需求。他们建议每张卡片上包含一些字段，例如需求原理、与其他需求的依赖关系、需求来源、支持性的材料等。这与图 4-13 中描述的结构化规格说明的例子中所使用的方法相似。

为了使用结构化方法刻画系统需求，需要定义一个或多个标准的需求模板，并将这些模板表示为结构化的表单。规格说明可以围绕系统操纵的对象、系统执行的功能或者系统处理的事件进行组织。图 4-13 中描述了一个基于表单的规格说明的例子，其中定义了当血糖处于安全区间时如何计算要供应的胰岛素剂量。

胰岛素泵 / 控制软件 / SRS/3.3.2

功能	计算胰岛素剂量：安全的血糖水平。
描述	当前测量的血糖水平在安全区间 3 ~ 7 个单位时，计算要供给的胰岛素的剂量。
输入	当前血糖读数 (r2)，前两个读数 (r0 和 r1)。
来源	当前读数来自传感器。其他读数来自存储。
输出	CompDose——要供给的胰岛素剂量。
目的地	主控制环。
动作	如果血糖水平稳定或在下降，或者虽然在升高但是上升率在下降，那么 CompDose 为 0。如果血糖水平在升高并且上升率也在增长，那么 CompDose 通过将当前血糖水平与前一血糖水平之差除以 4 并四舍五入来计算。如果结果为 0，那么将 CompDose 设为可以供应的最小的剂量（见图 4-14）。
要求	有前两个读数，这样可以计算血糖水平的变化率。
前置条件	胰岛素存储存有至少一个所允许的最大单剂量胰岛素。
后置条件	r0 被 r1 替代，而 r1 被 r2 替代。
副作用	无

图 4-13 胰岛素泵的需求结构化规格说明

当使用一种标准的格式来刻画功能性需求时，应当包含以下信息。

1. 对所刻画的功能或实体的描述；
2. 关于其输入以及输入来源的描述；

- 3. 关于其输出以及输出目的地的描述；
- 4. 关于计算所需的信息或者所需要的系统中其他实体的信息（“要求”部分）；
- 5. 关于所要采取的行动的描述；
- 6. 如果使用一个功能性的方法，那么用一个前置条件明确该功能调用前必须满足的条件，用一个后置条件刻画该功能调用后必须满足的条件；
- 7. 关于该操作的副作用（如果有的话）的描述。

使用结构化规格说明可以去除自然语言规格说明中的一些问题。规格说明中的可变性减少了，需求可以更有效地组织。然而，以一种清晰、无二义的方式书写需求有时候仍然很难，特别是当要刻画的计算很复杂时（例如，如何计算胰岛素剂量）。

针对这一问题，可以向自然语言需求中增加额外的信息，例如，通过使用表格或系统的图形化模型。这些可以显示计算是如何进行的，系统状态如何变化，用户如何与系统交互，以及动作序列如何执行。

当存在多种可能的情况并且需要描述每种情况下要采取的动作时，表格尤其有用。胰岛素泵中所需要的胰岛素用量的计算是基于血糖水平的变化率。变化率是使用当前和前面的读数来计算的。图 4-14 是一个关于如何使用血糖变化率来计算胰岛素供给量的表格化描述。

条 件	动 作
血糖水平在下降 ($r2 < r1$)	CompDose = 0
血糖水平稳定 ($r2 = r1$)	CompDose = 0
血糖水平在升高并且上升率在下降 ($(r2 - r1) < (r1 - r0)$)	CompDose = 0
血糖水平在升高并且上升率稳定或在增加 $r2 > r1$ & $((r2 - r1) \geq (r1 - r0))$	CompDose = round $((r2 - r1) / 4)$ 如果四舍五入后为 0，那么 CompDose = MinimumDose

图 4-14 胰岛素泵中的表格化计算规格说明

4.4.3 用况

用况（use case）是一种使用图形化模型和结构化文本描述用户与系统间交互的方式。用况最初是在 Objectory 方法（Jacobsen et al. 1993）中被引入，并且已经成为统一建模语言（UML）的一个基本特性了。按照最简单的用况形式，一个用况识别参与一个交互的参与者（actor），并且对交互的类型进行命名。在此基础上，可以添加一些描述与系统交互的附加信息。这些附加信息可以是一段文本描述，或者是一个或多个图形化模型，例如 UML 顺序图或状态图（见第 5 章）。

用况使用高层的用况图进行描述。一组用况的集合表示系统需求中将要描述的所有可能的交互。交互过程中的参与者（可以是人或其他系统）被表示为线条小人。每种类型的交互被表示为一个命名的椭圆。参与者和交互之间用线连接。作为一个选项，可以在线上增加箭头以表明交互是如何发起的。这些都在图 4-15 中有所描述，其中显示了一些来自 Mentcare 系统的用况。

用况识别系统与其用户或其他系统之间的各个交互。每个用况应当使用文本进行描述，然后与相应的 UML 模型（将更加详细地描述场景）联系起来。例如，图 4-15 中“安排会诊”

用况的简要描述可以如下面这样。

安排会诊允许两个或更多不同科室的医生来同时查看同一个病人的记录。一个医生通过从当前在线的医生的下拉菜单中选择参与人来发起会诊。病人记录显示在他们的屏幕上，但只有发起的医生可以编辑记录。此外，系统会创建一个文本聊天窗口来帮助进行相关行动的协调。这里的一个假设是，可以另外安排一个电话语音交流。

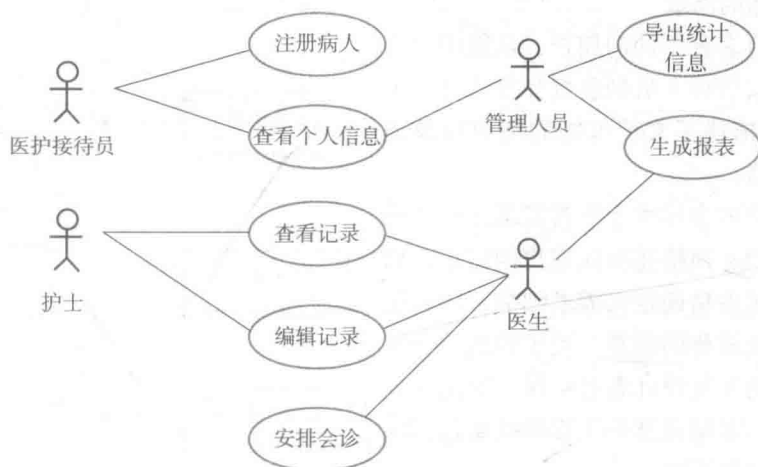


图 4-15 Mentcare 系统的用况

UML 是面向对象建模的标准，因此用况和基于用况的抽取在需求工程过程中有很多使用。然而，本书作者对于用况的经验是，它们粒度过细，因此难以用于讨论需求。利益相关者不理解用况这一名词；他们不觉得图形化模型很有用，并且经常对每一个系统交互的详细描述没什么兴趣。因此，与需求工程相比，我发现用况在系统设计中更有帮助。我将在第 5 章进一步讨论用况，其中描述了用况如何与其他系统模型一起用于描述系统设计。

有些人认为每个用况是一个单独的低层交互场景。而其他，例如 Stevens 和 Pooley (Stevens and Pooley 2006) 则认为每个用况包含一组相关的低层交互场景。其中每一个场景是一个对用况的单线程执行。因此，一个用况中包含一个代表常规交互的场景加上多个代表可能的异常情况的场景。在实践中，这两种理解都可以使用。

4.4.4 软件需求文档

软件需求文档，有时称为软件需求规格说明 (Software Requirement Specification, SRS)，是关于系统开发者应当实现的所有东西的正式陈述。需求文档可以同时包括一个系统的用户需求以及对于系统需求的详细规格说明。有时用户和系统需求被集成到同一个描述中。而有时，用户需求在系统需求规格说明的一个引言章节中描述。

当系统是外包开发的，即不同团队开发系统的不同部分，以及当详细的需求分析是必需的时候，需求文档十分关键。在其他情况下，例如，开发软件产品或业务系统时，详细的需求文档不一定是必需的。

敏捷方法认为需求变化非常之快，以至于需求文档在被写出来时就已经过时了，因此这

些工作都白费了。敏捷方法不使用正式的文档,而是经常增量地收集用户需求并且将它们作为简短的用户故事写在卡片或白板上。接下来,用户对这些故事在系统的下一个增量中实现的优先级进行排序。

对于需求不稳定的业务系统,这是个好办法,然而,书写一个定义系统的业务和可依赖性需求的简短支持文档仍然是有用的;当关注下一个系统发布的功能性需求时,很容易忘记适用于系统整体的需求。

需求文档有多种不同的用户,从组织中为系统付款的高层管理人员到负责开发软件的工程师。图 4-16 描述了文档可能的用户以及他们如何使用文档。

可能的用户的多样性意味着需求文档必须有所折中,其中必须描述面向客户的需求,面向开发者和测试者精确定义需求细节,同时包括关于未来系统演化的信息。关于预期的变更的信息可以帮助系统设计者避免做出限制性较强的设计决策,并帮助维护工程师按照新需求对系统进行适应性调整。

需求文档中应当包含的详细程度取决于所开发的系统类型以及所使用的开发过程。关键性系统需要详细的需求,因为必须详细分析安全性和信息安全以发现可能的需求错误。当系统将由其他独立的公司开发时(例如,通过外包),系统规格说明需要详细并且精确。如果使用内部的迭代化的开发过程,那么需求文档可以不那么详细。可以在系统开发过程中增加需求的细节并解决需求二义性。

图 4-17 描述了一个基于 IEEE 需求文档标准 (IEEE 1998) 的需求文档的组织结构。这个标准是通用的,可以根据特定的使用目的进行调整。此时,需要对标准进行扩展以包含与所预计的系统演化相关的信息。这些信息可以帮助系统的维护者,并且允许设计者考虑如何支持未来的系统特征。

很自然,需求文档中所包含的信息取决于所开发的软件类型以及要使用的开发方法。可以为一个包含由不同公司开发的硬件和软件的复杂工程化系统产生具有如图 4-17 中所示的结构的需求文档。这样的需求文档很可能会很长并且很详细。因此,在文档中包含一个全面的目录以及文档索引很重要,这样读者可以很容易找到他们需要的信息。

相比之下,面向一个内部开发的软件产品的需求文档可以去掉很多上面所建议的详细章节。其关注点将主要集中在定义用户需求以及高层的非功能性系统需求上。系统设计者和程序员根据自己的判断来决定如何满足系统的用户需求概要。

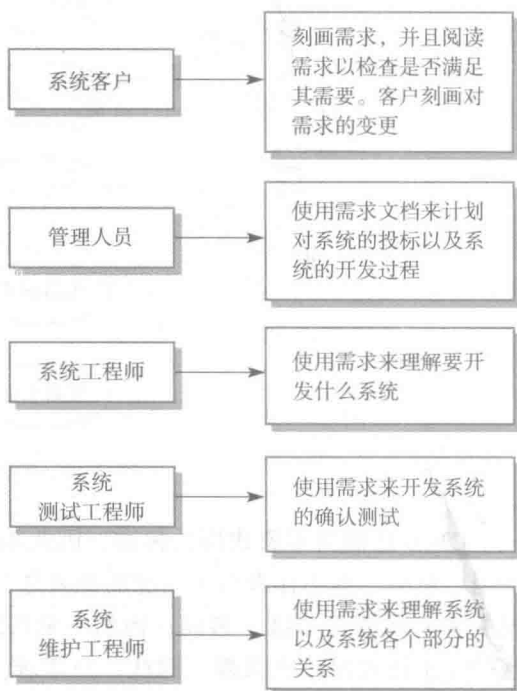


图 4-16 需求文档的用户

章	描 述
前言	这部分定义本文档所期望的读者人群，并且描述文档的版本历史，包括创建一个新版本的原因以及对于每个版本中所作修改的总结
引言	这部分描述系统的需要。其中应当简要描述系统的功能，并解释系统将如何与其他系统一起工作。还应当描述系统如何适应委托开发软件的组织的总体业务或战略目标
术语表	这部分定义了文档中所用的技术术语。不应该对读者的经验或专业知识进行假设
用户需求定义	这部分描述为用户提供的服务。非功能性系统需求也应当在这部分描述。这些描述使用客户可以理解的自然语言、图形或者其他表示法。必须遵循的产品和过程标准也应该在这里描述
系统体系结构	这部分描述所预计的系统体系结构的高层概览，显示各个系统模块上的功能分布。复用的体系结构构件应当进行强调
系统需求规格说明	这部分更详细地描述功能性需求和非功能性需求。如果有必要，可以向非功能性需求中增加进一步的细节。还可以定义与其他系统的接口
系统模型	这部分包括图形化的系统模型，显示系统构件之间以及系统及其环境之间的关系。可能的模型包括对象模型、数据流模型或语义数据模型
系统演化	这部分描述系统所基于的基本假设，以及所预计的由于硬件演化、用户需求变更等导致的变化。这部分对于系统设计者很有用，因为可以帮助他们避免做出会限制系统未来可能的变化的设计决策
附录	这部分提供与所开发的应用相关的详细、特定的信息，例如硬件和数据库描述。硬件需求定义了系统的最小配置和优化配置。数据库需求定义了系统所使用的数据的逻辑组织以及数据之间的关系
索引	可以包含几个文档索引。除了常规的字母序索引，还可以有图索引、功能索引等

图 4-17 需求文档的结构



需求文档标准

一些大型组织，例如美国国防部和 IEEE，已经定义了需求文档标准。这些标准通常都是通用的，但是作为开发更详细的组织标准的基础都是有用的。美国电气与电子工程师协会（Institute of Electrical and Electronic Engineers, IEEE）是最著名的标准提供者之一，他们已经开发了一个需求文档结构标准。该标准最适合于军事指挥和控制系统等生命周期很长且通常由多个组织一起开发的系统。

<http://software-engineering-book.com/web/requirements-standard/>

4.5 需求确认

需求确认是检查需求是否定义了客户真正想要的系统的过程。该过程与抽取和分析有重叠，因为它关注发现需求中的问题。需求确认非常重要，因为如果需求文档中的错误在开发过程中或在系统投入服务后被发现，则会导致广泛的返工开销。

通过进行系统变更修正一个需求问题的开销通常比修复设计或编码错误要高得多。一个需求变更通常意味着系统设计和实现也必须修改。而且，系统接下来还必须重新测试。

在需求确认过程中，应该对需求文档中的需求进行各种不同类型的检查。这些检查如下。

1. 正确性检查。检查需求是否反映了系统用户的真实需要。由于环境不断变化，用户需求可能在最初被抽取后已经发生了变化。
2. 一致性检查。文档中的需求不应该冲突。也就是说，不应该有相互矛盾的约束或者对同一个系统功能的不同描述。
3. 完整性检查。需求文档中的需求应当定义系统用户想要的所有功能以及约束。
4. 现实性检查。通过使用关于现有技术的知识，应当对需求进行检查以确保它们可以在系统预算范围内实现。这些检查应当考虑系统开发预算和进度。
5. 可验证性检查。为了减少客户和承包商之间可能的争议，所描述的系统需求应当总是可验证的。这意味着应当能够针对每一条所刻画的需求编写一组测试，以便显示所交付的系统满足该需求。



需求评审

需求评审的评委来自系统客户和系统开发团队，他们详细阅读需求文档并检查错误、异常和不一致的过程。一旦发现并记录了这些问题，接下来就需要客户和开发者协商所发现的问题应该如何解决。

<http://software-engineering-book.com/web/requirements-reviews/>

可以单独或结合使用一系列需求确认技术，例如下面这些。

1. 需求评审。由一个评审团队系统性地分析需求，检查错误和不一致性。
2. 原型化。包括开发一个可执行的系统模型，并与最终用户和客户一起使用该模型，来确认其是否满足他们的需要和期望。利益相关者对系统进行试验，并向开发团队反馈需求变更。
3. 测试用例生成。需求应该是可测试的。如果作为确认过程的一部分，设计针对一个需求的测试，那么经常可以揭示需求中的问题。如果一个测试很难或无法设计，这通常意味着需求难以实现并且应当重新考虑。在编写任何代码之前，根据用户需求开发测试是测试驱动的开发的一个重要组成部分。

不应当低估需求确认中所包含的问题。从根本上说，很难显示一组需求实际上满足用户的需要。用户需要想象系统运行以及系统将如何融入他们的工作。即使对于计算机专业技术人员也很难进行这种抽象的分析，对于系统用户就更难了。

由此造成的问题是，你很少能在需求确认过程中找到所有的需求问题。在相关利益相关者就需求文档已经达成共识后，不可避免地还需要通过进一步的需求变更来纠正遗漏和误解的需求。

4.6 需求变更

大型软件系统的需求总是在变化中。这样的频繁变化的一个原因是，这些系统经常被开发用于应对“非常规”的问题——无法完备定义的问题（Rittel and Webber 1973）。因为问题无法被充分定义，软件需求不可避免地会不完整。在软件开发过程中，利益相关者对于问

题的理解在不断变化(图 4-18),于是,系统需求必须演化以反映对这一变化的问题的理解。

一旦一个系统已经安装并被正常使用,新的需求不可避免地会出现。部分原因是原始需求中存在需要纠正的错误和遗漏。然而,大部分对系统需求的变更是由于系统的业务环境发生变化而产生的。

1. 系统的业务和技术环境总是会在系统安装后发生变化。可能引入新的硬件,或者更新已有的硬件;系统可能要与其他系统建立接口;业务优先级可能发生变化(所需要的系统支持会因此发生变化);新的法律和监管要求可能会出现并要求系统满足。

2. 为系统付钱的人和系统的用户经常是不同的人。系统客户由于组织和预算约束而提出需求。这些可能与最终用户的需求相互冲突,并且在交付后可能必须为用户增加新特征。

3. 大型系统通常有各种各样的利益相关者群体,他们的需求各不相同。他们的优先级可能相互冲突或矛盾。最终的系统需求不可避免地要进行折中,而有些利益相关者必须给予较高的优先级。根据经验,经常会发现对于给予不同利益相关者的支持平衡必须变化,而需求优先级要进行调整。

由于需求在演化中,需要对各个需求保持追踪并维护需求之间的依赖关系连接,这样就可以评价需求变更的影响。因此,需要一个正式的过程来提出变更并将它们与系统需求联系起来。这个“需求管理”过程应该在有了一个需求文档的草稿版本后尽快开始。

敏捷开发过程在设计时就考虑了如何应对需求在开发过程中发生变化。在这些过程中,当用户提出需求变更时,这一变化不需要经过正式的变更管理过程,而是由用户对变更进行优先级排序。如果一个变更请求优先级较高,那么要决定下一次迭代计划的哪些系统特征应当为实现该变更而暂时搁置。

该方法的问题是,用户不一定是决定需求变更的成本效益如何的最合适的人。在存在多个利益相关者的系统中,变更会使一些利益相关者受益,但对其他利益相关者则不然。此时,让一个可以平衡不同利益相关者需要的独立权威机构来决定变更是否应当被接受可能更好。

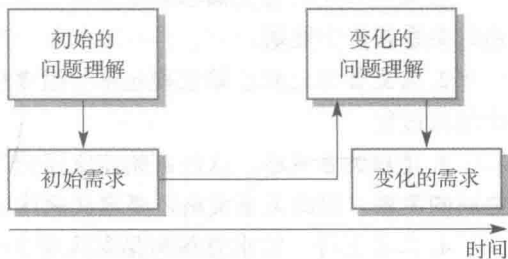


图 4-18 需求演化



稳定的需求 & 易变的需求

一些需求比其他需求更容易受到变更的影响。稳定的需求与组织中核心的、变化缓慢的活动相关。稳定的需求与基本的工作活动相关。易变的需求更容易变化,通常与反映组织如何完成工作而不是工作本身的支持性的活动相关。

<http://software-engineering-book.com/web/changing-requirements/>

4.6.1 需求管理计划

需求管理计划关注确定如何管理一组不断演化的需求。在计划阶段中,必须确定以下这

些问题。

1. 需求标识。每个需求必须被唯一标识,使得该需求可以被其他需求交叉引用,并且在追踪关系评价中使用。

2. 变更管理过程。该过程包括一组评估变更影响和成本的活动。后面的小节将更详细地介绍该过程。

3. 追踪关系策略。这些策略定义了应当记录的每个需求之间的关系以及需求与系统设计之间的关系。追踪关系策略还要定义这些记录应当如何维护。

4. 工具支持。需求管理包括对大量关于需求的信息的处理。可以使用的工具范围很广,从专业化的需求管理系统到共享的电子表格、简单的数据库系统。

需求管理需要自动化的支持,相关的软件工具应当在这个计划阶段中选择好。工具需要支持以下这些方面。

1. 需求存储。需求应当在一个安全、受管理,而且参与需求工程过程的每个人都能访问的数据库中进行维护。

2. 变更管理。如果有活跃的工具支持,那么变更管理过程(见图4-19)可以简化。工具可以对所提出的变更以及对这些提议的回应保持追踪。

3. 追踪关系管理。如上所述,追踪关系的工具支持使得开发组可以发现相关的需求。有些可用的工具使用自然语言处理技术来帮助发现需求之间可能的关系。

对于小系统,不需要使用专门的需求管理工具。需求管理可以使用共享的 Web 文档、电子表格和数据库来支持。然而,对于更大的系统,应该使用 DOORS (IBM 2013) 这样的系统专业化工具支持,可以使追踪大量的变化需求变得容易得多。

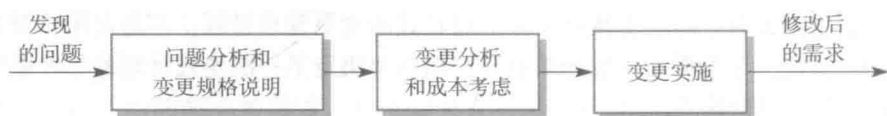


图 4-19 需求变更管理



需求追踪关系

需要对需求、需求来源以及系统设计之间的关系保持追踪,这样就可以针对所提出的变更分析其原因,分析变更很可能对于系统其他部分产生的影响。开发人员要能够追踪一个变更的影响是如何像波纹一样在整个系统中传播的。为什么?

<http://software-engineering-book.com/web/traceability/>

4.6.2 需求变更管理

需求变更管理(见图4-19)针对的是需求文档被批准后对系统需求所提出的所有变更。变更管理很重要,因为要决定实现新需求的成本和收益是否合算。使用正式的变更管理过程的好处是,所有的变更请求都可以以一致的方式进行处理,而对于需求文档的修改可以以一种受控的方式进行。

变更管理过程包含下面 3 个主要阶段。

1. 问题分析和变更规格说明。变更管理过程开始于所发现的一个需求问题, 或者有时开始于一个特定的变更请求。在此阶段中, 对问题或变更请求进行分析以检查其是否合理。该分析被反馈给变更请求者, 然后请求者会回应一个更加特定的需求变更请求, 或者决定撤回请求。

2. 变更分析和成本考虑。所提出的变更的效果使用追踪关系信息以及关于系统需求的基本知识进行评估。根据对需求文档以及系统设计和实现的修改, 对实施该变更的成本进行估算。该分析一旦完成, 就可以做出一个是否继续处理该需求变更的决定。

3. 变更实施。如果有必要, 还需要对需求文档、系统设计和实现进行修改。应当对需求文档进行适当组织, 以便在修改文档时不需要进行大范围的重写或重新组织。与程序一样, 文档的可修改性是通过尽量减少外部引用、让文档的各个部分尽量模块化等手段实现的。这样, 单个部分的修改和替换不会影响文档的其他部分。

如果一个新需求必须被紧急实现, 那么总是存在一种倾向, 即先修改系统, 然后再找时间修改需求文档。这种做法几乎总是会不可避免地导致需求规格说明和系统实现不同步调。一旦对系统进行了修改, 很容易忘记相应地修改需求文档。在有些情况下, 紧急的系统变更必须实施。此时, 尽快更新需求文档以包含修改后的需求是很重要的。

要点

- 软件系统的需求明确了系统应该做什么, 并定义了对于系统运行和实现的约束。
- 功能性需求是对系统必须提供的服务的陈述或者对一些计算必须如何进行的描述。
- 非功能性需求经常约束所开发的系统以及所使用的开发过程。这些可以是产品需求、组织需求或外部需求。它们经常与系统的涌现性特性相关, 因此适用于系统整体。
- 需求工程过程包括需求抽取、需求规格说明、需求确认、需求管理。
- 需求抽取是一个可以表示为一个活动螺旋的迭代化过程, 其中的活动包括需求发现、需求分类和组织、需求协商、需求文档化。
- 需求规格说明是正式地描述用户系统需求并创建一个软件需求文档的过程。
- 软件需求文档是对于系统需求的一个达成共识的陈述。应当对该文档进行组织, 以便系统客户和软件开发都可以使用它。
- 需求确认是检查需求的正确性、一致性、完整性、现实性、可验证性的过程。
- 业务、组织和技术变化会不可避免地导致软件系统需求的变化。需求管理是管理并控制这些变化的过程。

阅读推荐

《Integrated Requirements Engineering: A Tutorial》是一篇教程式的论文, 其中讨论了需求工程活动, 以及如何对它们进行适应性调整以适应现代的软件工程实践。(I. Sommerville, IEEE Software, 22 (1), January-February 2005) <http://dx.doi.org/10.1109/MS.2005.13>

《Research Directions in Requirements Engineering》是一个很好的对需求工程的调研, 其中突出了在应对规模和敏捷性等问题方面的未来研究挑战。(B. H. C. Cheng and J. M. Atlee, Proc. Conf. on Future of Software Engineering, IEEE Computer Society, 2007) <http://dx.doi.org/10.1109/FOSE.2007.17>

《Mastering the Requirements Process》(第3版)是一本写得很好、很容易理解的书,其中基于一个特定的方法(VOLERE),但是也包含很多关于需求工程的好的一般化建议。(S. Robertson and J. Robertson, 2013, Addison-Wesley)

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap4/>

支持视频的链接: <http://software-engineering-book.com/videos/requirements-and-design/>

胰岛素泵系统需求文档: <http://software-engineering-book.com/case-studies/insulin-pump/>

Mentcare 系统需求信息: <http://software-engineering-book.com/case-studies/mentcare-system/>

练习

- 4.1 识别并简要描述 4 种可能会由基于计算机的系统描述的需求。
- 4.2 找出下面这段售票系统需求陈述中有二义或遗漏的地方:
一个自动化的售票机销售火车票。用户选择他们的目的地并输入信用卡和个人身份信息。机器吐出火车票,而用户的信用卡账户会进行付款。当用户按下启动按钮,一个显示候选目的地的菜单被激活,同时系统向用户显示一条选择目的地以及所需要的票的类型的消息。一旦选择了目的地,系统显示票价并请客户输入他们的信用卡。检查信用卡是否有效之后,系统请用户输入个人身份(PIN 码)。信用卡交易确认后,票被吐出。
- 4.3 使用本章中介绍的结构化方法重写以上陈述。以合理的方式解决所识别的二义性。
- 4.4 为售票系统书写一组非功能性需求,明确所期望的可靠性和响应时间。
- 4.5 使用本章所建议的技术(其中自然语言描述呈现为标准的格式),针对下面这些功能书写看似合理的用户需求。
 - 一个无人值守的汽油泵系统,包含一个信用卡读卡器。客户通过读卡器刷卡,然后输入所需要的汽油量。系统供应相应数量的汽油,并从客户的账户扣款。
 - 一个银行 ATM 机的现金取款功能。
 - 在一个互联网银行系统中,允许客户从所持有的当前银行的一个账户中转账到同一银行的另一个账户的功能。
- 4.6 一个负责起草系统需求规格说明的工程师,应当如何记录功能性需求和非功能性需求之间的关系?请给出你的建议。
- 4.7 根据你自己关于 ATM 机使用的经验,开发一组可以作为理解 ATM 机系统需求的基础的用况。
- 4.8 哪些人应该参加需求评审?画一个过程模型,描述需求评审如何组织。
- 4.9 当系统面临必须满足的紧急修改时,系统中的软件可能不得不在相应的需求变更被批准前就进行修改。建议一个实施这类修改的过程模型,使之可以确保需求文档和系统实现不会变得不一致。
- 4.10 假设你刚在一个软件用户那里获得一份工作,该用户单位正好与你先前的雇主单位签订了一份系统开发合同。你发现你现在的单位与先前的雇主单位对需求有不同的理解。讨论在这种情况下你应该怎样做。你知道如果需求中的二义性不解决那么你现在所在的单位的成本将会增加。然而,你同时也对先前的雇主单位有相应的保密责任。

参考文献

- Crabtree, A. 2003. *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.
- Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.
- IBM. 2013. "Rational Doors Next Generation: Requirements Engineering for Complex Systems." <https://jazz.net/products/rational-doors-next-generation/>
- IEEE. 1998. "IEEE Recommended Practice for Software Requirements Specifications." In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press.
- Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.
- Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.
- Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.
- Robertson, S., and J. Robertson. 2013. *Mastering the Requirements Process, 3rd ed.* Boston: Addison-Wesley.
- Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twidale. 1993. "Integrating Ethnography into the Requirements Engineering Process." In *RE'93*, 165–173. San Diego, CA: IEEE Computer Society Press. doi:10.1109/ISRE.1993.324821.
- Stevens, P., and R. Pooley. 2006. *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison-Wesley.
- Suchman, L. 1983. "Office Procedures as Practical Action: Models of Work and System Design." *ACM Transactions on Office Information Systems* 1 (3): 320–328. doi:10.1145/357442.357445.
- Viller, S., and I. Sommerville. 2000. "Ethnographically Informed Analysis for Software Engineers." *Int. J. of Human-Computer Studies* 53 (1): 169–196. doi:10.1006/ijhc.2000.0370.

系统建模

目标

本章的目标是介绍可以作为需求工程和系统设计过程的一部分开发的系统模型。阅读完本章后，你将：

- 理解图形化模型可以如何用来表示软件系统，以及为什么需要多种不同的模型来完整表示一个系统；
- 理解几个基本的系统建模视角，包括上下文、交互、结构、行为；
- 理解主要的统一建模语言（UML）图类型，以及如何在系统建模中使用这些图；
- 了解模型驱动的工程，其中会从结构以及行为模型自动生成一个可执行的系统。

系统建模就是建立系统抽象模型的过程，其中每一个模型表示系统的一个不同的视角或观点。系统建模现在通常意味着在 UML 中的图类型基础上使用某种图形化的表示法表示系统。然而，也有可能要开发系统的形式化（数学）模型，通常将其作为详细的系统规格说明。本章将介绍基于 UML 的图形化建模，形式化建模将在第 10 章中简要介绍。

在需求工程过程中使用模型，是为了帮助得到详细的系统需求；在设计过程中使用模型，是为了向实现系统的工程师描述系统；在实现系统之后还要使用模型，是为了描述系统的结构和运行。可以同时针对现有系统和待开发系统开发系统模型。

1. 现有系统的模型在需求工程过程中使用。这些模型帮助阐明现有系统做什么，并且可以用于让利益相关者之间的讨论聚焦于当前系统的优势和弱点。

2. 新系统的模型在需求工程过程中使用。这些模型帮助解释对其他系统利益相关者所提出的需求。工程师使用这些模型来讨论设计方案并描述系统以用于实现。如果你使用模型驱动的工程化过程（Brambilla, Cabot, and Wimmer 2012），那么你可以在系统模型基础上生成一个完整的或部分系统实现。



统一建模语言

统一建模语言（The Unified Modeling Language, UML）是一组 13 种不同的图形类型，它们可以被用于建模软件系统。UML 是在 20 世纪 90 年代的面向对象建模方面的工作基础上出现的，其中相似的面向对象表示法被集成到一起创建了 UML。UML 的一个重大修改（UML 2）于 2004 年定稿。UML 被广泛接受成为开发软件系统模型的标准方法。还有一些 UML 的变体（例如，SysML）被提出，用于更通用的系统建模。

<http://software-engineering-book.com/web/uml/>

理解一个系统模型并不是系统的一个完备表示，这一点很重要。系统模型有意去掉一些细节以使模型更容易理解。模型是所研究的系统的一种抽象，而不是系统的另一种表示。系

统的一个表示应当包含关于所表示的实体的所有信息。而一个抽象则有意简化一个系统设计并选取最显著的特性。例如,本书附带的 PowerPoint 是对本书要点的抽象。然而,如果将本书从英语翻译为意大利语,那么将产生另一种表示。翻译者的意图是尽量保持本书在英文版中的所有信息。

你可以开发不同的模型来从不同的视角表示系统。

1. 外部视角,会对系统的上下文或环境进行建模;
2. 交互视角,会对系统及其环境或者系统的构件之间的交互进行建模;
3. 结构化视角,会对系统的组织或者系统所处理的数据的结构进行建模;
4. 行为视角,会对系统的动态行为以及系统如何响应事件进行建模。

当开发系统模型时,开发者经常可以灵活决定图形化表示法的使用方式。开发者并不总是需要严格坚持一些细节。一个模型的细节和严格性取决于开发者打算如何使用它。以下是图形化模型的3种常见的使用方式。

1. 作为一种推动关于现有或所设想的系统的讨论以及使讨论聚焦的方式。模型的目的是推动以及聚焦参与系统开发的软件工程师之间的讨论。模型可以不完整(只要它们覆盖了讨论的要点),而且可能会以一种非正式的方式使用建模表示法。这是敏捷建模中常见的模型使用方式(Ambler and Jeffries 2002)。

2. 作为一种文档化现有系统的方式。当模型被用于文档化时,它们不需要是完整的,因为你可能只需要使用模型去描述系统的一些部分。然而,这些模型必须是正确的——它们应当正确地使用相应的建模表示法,并且是对系统的准确描述。

3. 作为一种可以用于生成系统实现的详细系统描述。当模型作为基于模型的开发过程的一部分被使用时,系统模型必须是完整而且正确的。它们可以作为生成系统源代码的基础,因此必须非常小心不要混淆相似但含义各不相同的符号(例如,线形箭头和块状箭头)。

在本章中,将使用统一建模语言(UML)(Rumbaugh, Jacobson, and Booch 2004; Booch, Rumbaugh, and Jacobson 2005)中所定义的图形。UML已经成为面向对象建模的标准语言。UML有13种图的类型,因此支持许多不同类型的系统模型的创建。然而,一些调研(Erickson and Siau 2007)表明,大多数UML用户认为5种类型的图就可以表示系统的基本特性了。因此,本章关注以下5种UML图类型。

1. 活动图(activity diagram),描述一个过程或数据处理中所包含的活动。
2. 用况图(use case diagram),描述一个系统与其环境之间的交互。
3. 顺序图(sequence diagram),描述参与者与系统之间以及系统构件之间的交互。
4. 类图(class diagram),描述系统中的对象类以及这些类之间的联系。
5. 状态图(state diagram),描述系统如何对内部和外部的的事件做出响应。

5.1 上下文模型

在系统规格说明的早期阶段,你应当确定系统的边界,也就是说确定哪些属于、哪些不属于所开发的系统。这其中包含与系统利益相关者一起工作来决定哪些功能应当包含在系统中,以及哪些处理和操作应当在系统的运行环境中执行。开发者可能会决定在所开发的软件中应当实现对一些业务过程的自动化支持,但是其他一些过程则应该手工完成或由其他系统来支持。应该考虑所开发的系统在功能上与已有系统可能存在的重叠部分,并决定新功能应当在哪里实现。这些决定应当在该过程的早期做出,以控制理解系统需求和设计所需要的系

统成本和时间。

有些时候,系统及其环境的边界相对清楚。例如,当一个自动化系统准备取代一个已有的手工或计算机化的系统时,新系统的环境通常与现有系统的环境一样。而在其他情况下则存在更多的灵活性,你需要在需求工程过程中确定系统及其环境的边界由哪些因素构成。

例如,假设你正在为 Mentcare 病人信息系统开发规格说明。该系统将会管理关于参加心理健康诊断以及安排治疗的病人的信息。在为该系统开发规格说明时,必须确定系统是否应当只关注收集诊疗的信息(使用其他系统来收集病人的个人信息),还是系统也应该收集病人的个人信息。依赖于其他系统获取病人信息的好处是避免了数据重复。然而主要的不利之处是使用其他系统可能会让信息访问比较慢,另外如果这些系统不可用那么 Mentcare 系统也会无法使用。

在有些情况下,系统的用户基础非常分散,用户有很多不同的系统需求。你可能会决定不去定义系统边界,而是开发一个可以通过调整适应不同用户需要的可配置系统。我们在第1章中所介绍的 iLearn 系统的开发中就采用了这种方法。该系统的用户范围很广,从很小的还无法阅读的小孩子到年轻人、他们的老师以及学校管理人员。因为这些用户群体需要不同的系统边界,我们刻画了一个可以允许系统在部署时再确定边界的可配置系统。

系统边界的定义不是一个与价值无关的判断。社会和组织关注点可能意味着系统边界的位置可能要由非技术因素决定。例如,人们可能会有意划定一个系统的边界,以使完整的分析过程可以在一个地点进行;可以通过选择边界来避免请教一个特别不好相处的管理人员;可以通过选择边界使系统成本增加,并且系统开发部门因此必须扩张以完成系统的设计和实现。

一旦关于系统边界的一些决定已经做出,需要进行的一部分分析活动是定义系统上下文以及系统对于其环境的依赖。通常,产生一个简单的体系结构模型是该活动的第一步。

图 5-1 是一个描述 Mentcare 系统及其环境中的其他系统的上下文模型。从中可以看到 Mentcare 系统与一个预约系统(Appointments system)以及一个更通用的病人记录系统(Patient record system)相连接并共享数据。该系统还与管理报表系统(Management reporting system)、医院许可系统(Admissions system)以及一个为研究目的收集信息的统计系统(HC statistics system)相连接。最终,该系统使用一个处方系统(Prescription system)来生成病人的药物处方。

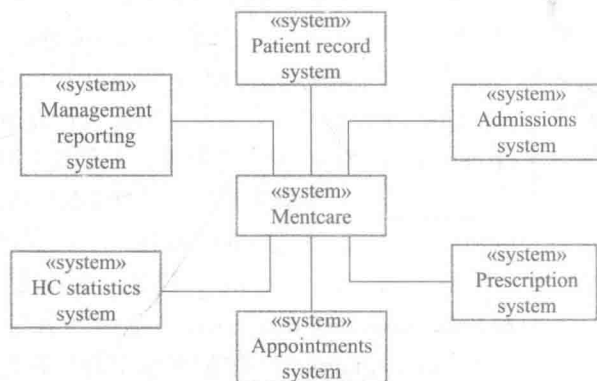


图 5-1 Mentcare 系统的上下文

上下文模型通常显示环境包括多个其他的自动化系统。然而,这种模型没有显示环境中的系统与所描述的系统之间关系的类型。外部系统可能会产生提供给系统的数据或者消费来自系统的数据。它们可能与系统共享数据,它们可能会直接通过网络连接或完全没有连接。这些外部系统可能在物理上位于同一位置或者位于不同的建筑中。所有这些关系都可能影响所定义的系统的需求和设计,必须加以考虑。因此,简单的上下文模型可以和其他模型(例如,业务过程模型)一起使用。业务过程模型描述了人以及自动化的过程,其中会使用特定的软件系统。

UML 活动图可以用于显示系统使用所处的业务过程。图 5-2 是一个 UML 活动图，其中描述了 Mentcare 系统如何用于一种重要的心理健康问题治疗过程——强制留置。

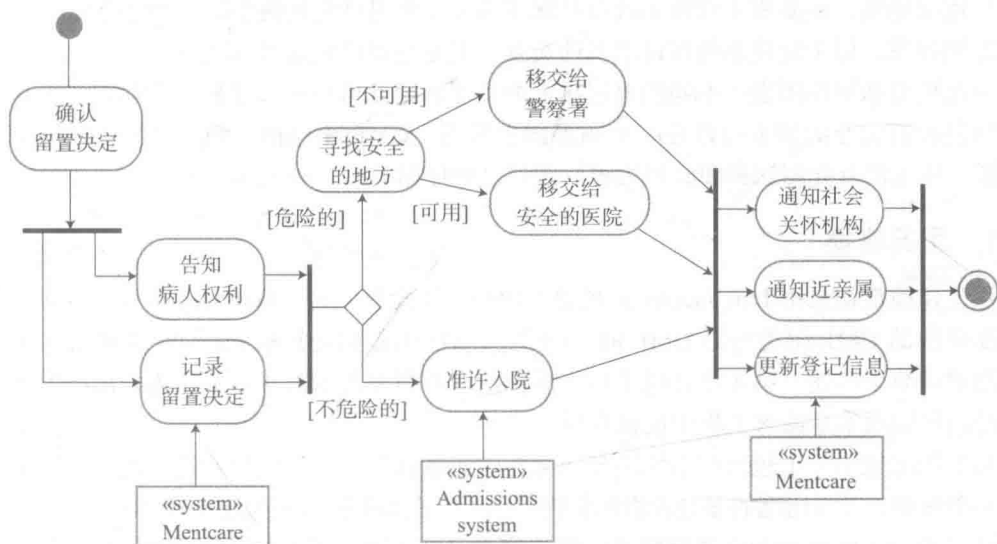


图 5-2 强制留置的过程模型

有时候，患有心理健康问题的病人可能对他人或自己造成危险。因此可能必须违背他们的意愿强行将他们留置在医院，以使处置过程可以得到监管。这种留置需要遵循严格的法律保护，例如，留置病人的决定必须进行常规评审以避免在没有合理原因的情况下长期留置病人。Mentcare 系统的一个关键功能是确保这种保护得到实现，并且病人的权利能够得到尊重。

UML 活动图显示了过程中的活动以及从一个活动到另一个活动的控制流。活动的开始用一个实心圆表示，结束用一个在另一个圆里面的实心圆表示。圆角矩形表示活动，也就是必须执行的特定的子过程。还可以在活动图中包含对象。图 5-2 显示的相关系统可用来支持强制留置过程中的不同子过程。图中通过使用 UML 的构造型（stereotype）特性显示了这些都是独立的系统，其中方框中 «» 括起来的是实体的类型。

箭头表示从一个活动到另一个活动的工作流，实心条（粗线段）表示活动间的协调。当来自多个活动的流到达一个实心条时，表示所有这些活动结束之后才能继续进行。当从一个实心条流出多个活动时，表示这些活动可以并行执行。因此，图 5-2 中显示，通知社会关怀机构和病人的近亲属，以及更新留置登记信息可以同时进行。

可以在箭头旁标注警戒条件（方括号中），说明什么情况下过程流可以进行。从图 5-2 中可以看到，对社会有危险和没有危险的病人的不同的流。对社会有危险的病人必须留置在一个安全的地方。然而，有自杀倾向、会对自身产生危险的病人可以准许进入医院中合适的病房，这些地方可以让病人处于严密监视之中。

5.2 交互模型

所有系统都包含某种类型的交互。可以是用户交互，其中包含用户输入和输出；也可以是所开发的软件和环境中的其他系统之间的交互；或者是软件系统内部不同构件之间的交互。用户交互建模很重要，因为它可以帮助识别用户需求。建模系统间的交互可以突出可能出现的通信问题。建模构件交互可以帮助我们理解所提出的系统结构是否能实现所要求的系

统性能和可依赖性。

本章讨论以下两个相关的交互建模方法：

- 1. 用况建模，主要用于建模系统与外部主体（人类用户或其他系统）之间的交互；
- 2. 顺序图，用于建模系统构件之间的交互，但是也可以包含外部主体。

用况模型和顺序图表示不同抽象层次上的交互，因此可以一起使用。例如，一个高层用况中所包含的交互的细节可以在一个顺序图中描述。UML 还包括一种可以用于建模交互的通信图。这里没有介绍这种图，因为通信图只是顺序图的另一种表示。

5.2.1 用况建模

用况建模最初是由 Ivar Jacobsen 在 20 世纪 90 年代开发的（Jacobsen et al. 1993），而支持用况建模的 UML 图类型是 UML 的一部分。一个用况可以作为一个用户在该交互中对系统的期望的简单描述。第 4 章介绍了用于需求抽取的用况。如第 4 章中所述，用况模型在系统设计的早期而不是需求工程中比较有用。

每个用况表示一个包含与系统的外部交互的离散任务。如果使用最简单的形式，用况表示为一个椭圆，参与用况的参与者表示为线形小人。图 5-3 展示了 Mentcare 系统的一个用况，其中描述了一个从 Mentcare 系统向更通用的病人记录系统上传数据的任务。这个更通用的系统保存了关于病人的总结数据，而不是每次诊疗的数据（这些数据保存在 Mentcare 系统中）。



图 5-3 “传输数据”用况

注意该用况中有两个参与者——一个传输数据的操作人员，病人记录系统。线形小人表示法最初使用时是为了表示与人的交互，但现在也可以表示其他外部系统和硬件。按照规范的画法，用况图应该使用没有箭头的线，因为在 UML 中箭头表示消息流的方向。很明显，在一个用况中消息会双向传递。然而，图 5-3 中的箭头以一种非正规的方式表示医疗接待员发起事务以及数据是传输到病人记录系统中。

用况图对于交互给出了一个简单的概览，在此基础上还要增加更多的细节以获得完整的交互描述。这些细节可以是简单的文本描述、使用表格的结构化描述，或者顺序图。可以根据具体的用况以及所需要的详细程度从中选择最合适的格式。我发现标准的表格化格式是最有用的。图 5-4 显示了一个“传输数据”用况的表格化描述。

Mentcare 系统：传输数据	
参与者	医疗接待员、病人记录系统（PRS）
描述	医疗接待员可以从 Mentcare 系统向健康管理机构维护的通用的病人记录数据库传输数据。所传输的信息可以是更新的个人信息（地址、电话号码等）或者病人的诊断和治疗情况总结
数据	病人个人信息、治疗总结
触发激励	医疗接待员发出的用户命令
响应	PRS 已经更新的确认信息
注解	医疗接待员必须具有访问病人信息以及 PRS 的适当的信息安全许可

图 5-4 “传输数据”用况的表格化描述

复合的用况图显示了多个不同的用况。有时候可以只使用一个复合用况图包含系统中所有可能的交互。然而，如果用况数量较多那么就无法做到了。在这种情况下，可以开发多个用况图，其中每一个显示相互关联的用况。例如，图 5-5 显示了 Mentcare 系统所有的用况，其中包含参与者“医疗接待员”。在此基础上应该为其中每一个用况提供更加详细的描述。

UML 包含一系列元素来支持将一个用况的全部或部分共享给其他用况图。虽然这些元素有时候对系统设计者有一定帮助，我的经验是很多人（特别是最终用户）发现他们很难理解这些元素。为此，这里没有描述这些元素。

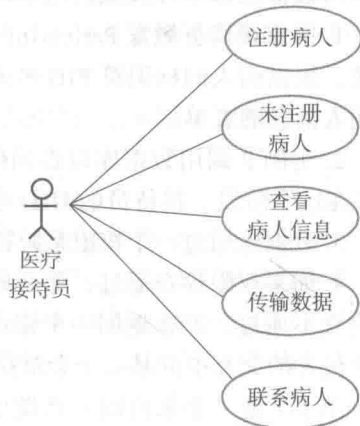


图 5-5 包含“医疗接待员”角色的用况

5.2.2 顺序图

UML 中的顺序图主要用于建模参与者与系统中的对象之间的交互以及这些对象自身相互间的交互。UML 为顺序图提供了丰富的语法，这使得顺序图可以对许多不同种类的交互进行建模。由于篇幅限制这里无法介绍所有可能的描述方式，因此，这里介绍的重点是顺序图的一些基本特性。

顾名思义，顺序图显示在一个特定的用况或用况实例执行过程中发生的交互序列。图 5-6 是一个描述顺序图的基本表示法的例子。这个图对查看病人信息用况中所包含的交互进行了建模，其中一个医疗接待员可以看到一些病人信息。

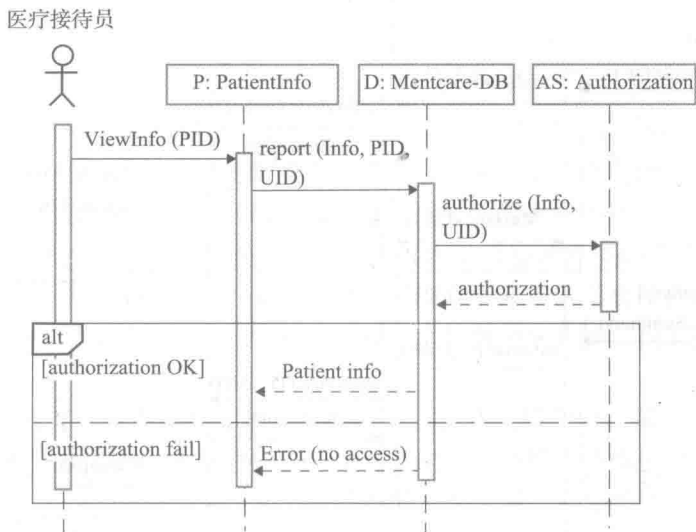


图 5-6 查看病人信息的顺序图

该顺序图的顶部列出了参与交互的对象和参与者，每个下面还有一个垂直方向的虚线。带标签的箭头表示对象之间的交互。虚线上的矩形表示所关心的对象的生命线（即对象实例参与计算的时间）。可以按照从上到下的顺序阅读这些交互。箭头上的标签表示对对象的调用、调用参数以及返回值。这个例子中还显示了用于表示可选项的表示法。其中包括带有

alt 名称的方框、方括号中的条件、用虚线分隔的可选的交互选项。

可以按照以下方式理解图 5-6。

1. 医疗接待员触发 PatientInfo(病人信息) 对象类的一个实例 P 中的 ViewInfo(查看信息) 方法, 提供病人的标识符 PID 来确定所需要的信息。P 是一个用户界面对象, 呈现为一个显示病人信息的表单。

2. 实例 P 调用数据库以返回所需要的信息, 提供接待员的标识符 UID 用于信息安全检查(在这个阶段, 接待员的 UID 来自哪里不重要)。

3. 数据库通过一个权限系统检查接待员是否具有执行这个动作的权限。

4. 如果权限检查通过, 那么病人信息被返回并呈现在用户屏幕上的一个表单上。如果权限检查不通过, 那么返回一个错误消息。左上角那个带有 alt 标签的方框是一个选择框, 表示所包含的交互中的某一个会被执行。选择每个选项的条件显示在方括号中。

图 5-7 是一个来自同一系统的顺序图的进一步的例子。其中展示了两个新的特性, 包括系统中参与者之间的直接通信以及作为操作序列一部分的对象的创建。在这个例子中, Summary(总结) 类型的一个对象被创建用于封装总结数据, 这些数据将被上传到一个国家 PRS(病人记录系统) 中。可以按照下面的方式理解这幅图。

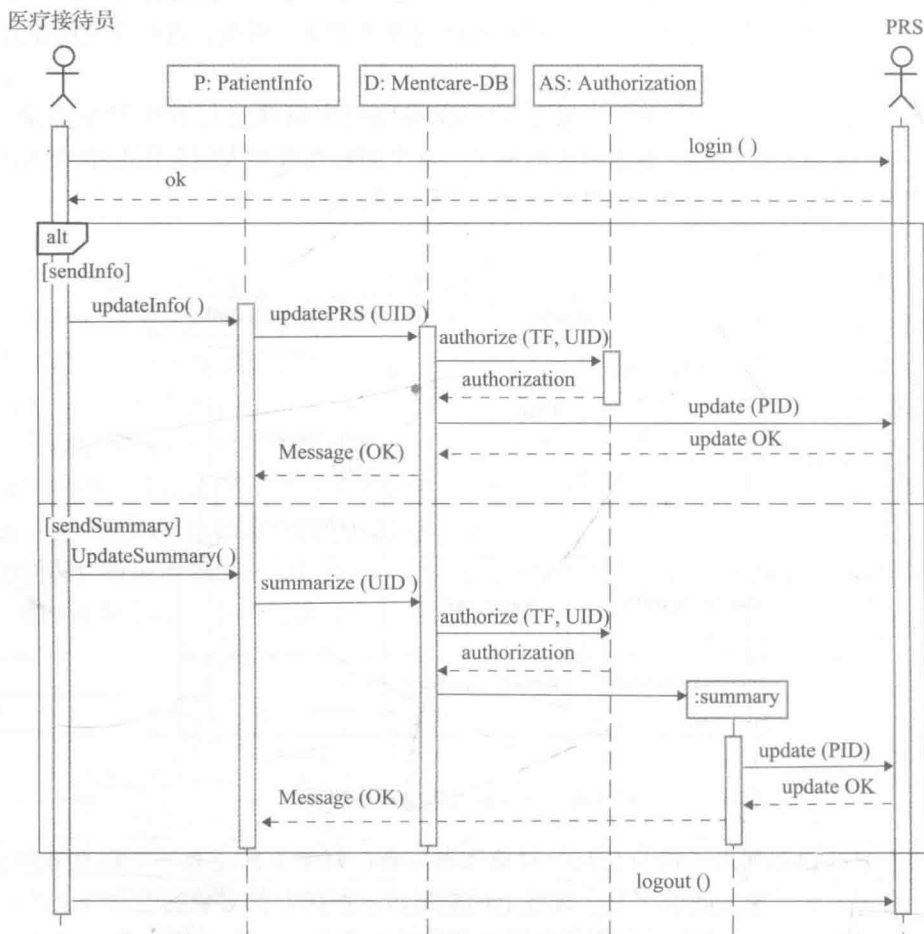


图 5-7 传输数据的顺序图

1. 接待员登录到 PRS 上。
2. 系统提供两个选项（如 alt 框所示）。这两个选项分别是：直接传输来自 Mentcare 数据库的更新后的病人信息到 PRS 中；传输来自 Mentcare 数据库的健康数据总结到 PRS 中。
3. 在每个选项中，接待员的访问许可都要通过权限系统进行检查。
4. 个人信息可以直接从用户界面对象传输到 PRS。而总结记录则要从数据库中读取数据后创建，然后传输该记录。
5. 传输完成后，PRS 发出一个状态消息，然后用户登出系统。

除非将顺序图用于代码生成或详细的文档化，否则不需要在这些图中包含所有交互。如果在开发过程早期开发系统模型来支持需求工程和高层设计，那么有许多交互都取决于实现决策。例如，在图 5-7 中，关于如何获取用户标识符以进行权限检查的决策就可以后面再决定。在某个实现中，这一点可能包含与 User（用户）对象的交互。由于这一点在当前阶段不重要，因此不需要将其包含在顺序图中。

5.3 结构模型

软件的结构模型按照构成系统的构件以及它们之间的关系显示系统的组织。结构模型可以是描述系统设计组织的静态模型，也可以是描述系统执行时的组织的动态模型。这两种模型是不一样的——系统的动态组织是一组相互交互的线程，这与系统构件的静态模型很不一样。

可以在讨论和设计系统体系结构时创建系统的结构模型。这些模型可以是总体的系统体系结构模型或者更加详细的系统中的对象及其关系的模型。

本章关注使用类图对软件系统中的对象类的静态结构进行建模。体系结构设计是软件工程中的一个重要问题，UML 构件图、包图、部署图都可以用于描述体系结构模型。本书第 6 章和第 17 章将介绍体系结构建模。

5.3.1 类图

当开发一个面向对象系统模型来显示系统中的类以及类之间的关联时，可以使用类图。大致上可以将对象类理解为一类系统对象的泛化定义。关联是类之间的链接，表示这些类之间存在某种关系。因此，每个类可能都要具有与之相关联的类的一些知识。

在软件工程过程的早期阶段开发模型时，对象表示现实世界中的一些东西，例如病人、处方、医生。随着软件实现被开发出来，要定义实现对象来表示系统所操纵的数据。本章关注作为需求或者软件设计过程早期的一部分的现实世界对象的建模。相似的方法也被用于数据结构建模。

UML 中的类图可以在不同的详细程度上表达。在开发模型时，第一个阶段通常是看一看现实世界、识别基本的对象并将它们表示为类。这些图最简单的描述方法是在一个方框中写上类名，还可以在类之间画一条线来表示存在的关联关系。例如，图 5-8 是一个简单的类图，其中显示了两个类——Patient（病人）和 Patient Record（病人记录），以及它们之间的关联关系。在这个阶段，不需要说明这个关联关系是什么。



图 5-8 UML 类和关联关系

图 5-9 进一步发展了图 5-8 中的简单类图，以显示 Patient 类的对象还参与了一系列其他类之间的关系。这个例子中显示了对于关联关系的命名，使读者可以知道所存在的关系的类型。

图 5-8 和图 5-9 显示了类图的一个重要特性——显示有多少对象参与关联关系的能力。图 5-8 中关联关系的两端都标注了 1，意味着这些类的对象之间存在一对一的关系，也就是说每个病人有且仅有一个记录，而每个记录保存一个病人的信息。

从图 5-9 中可以看到还有其他关于重数的表达。可以精确定义参与关系的对象的数量（例如 1..4），或者使用一个星号（*）来表示参与关联关系的对象数量不确定。例如，图 5-9 中 Patient（病人）和 Condition（症状）之间的关系的重数（1..*）表示一个病人可能存在多种症状，而同一个症状可能与多个病人相关。

在这种详细程度上，类图看起来像语义数据模型。语义数据模型用于数据库设计中。它们描述数据实体、与它们相关的属性，以及这些实体之间的关系（Hull and King 1987）。UML 没有针对数据库设计的图类型，因为 UML 使用对象以及它们的关系来进行数据建模。然而，可以使用 UML 来表示语义数据模型。可以认为语义数据模型中的实体是简化的对象类（它们没有操作），属性是对象类的属性，而关系是对象类之间命名的关联关系。

当显示类之间的关联关系时，最好以最简单的不包含属性或操作的方式来表示这些类。为了更加详细地定义对象，可以增加关于它们的属性（对象的特征）和操作（对象的功能）的信息。例如，一个 Patient 对象具有属性 Address（地址），还可以包含一个名为 ChangeAddress 的操作，在一个病人表明他刚从一个地址搬到另一个地址时调用该操作。

UML 通过扩展表示类的简单矩形来显示属性和操作。图 5-10 中描述了这些，其中显示了一个表示医生和病人之间的诊疗的对象。

1. 对象类的名称在顶部。
2. 类属性在中间，其中包括属性名以及可选的属性类型。图 5-10 中没有显示类型。
3. 与对象类相关的操作（在 Java 和其他面向对象语言中称为“方法”）在矩形中最底下的部分。图 5-10 中显示了一部分但不是所有的操作。

图 5-10 中所显示的例子中，假设医生记录语音笔记，这些笔记此后会被转录以记录诊疗的细节。为了开处方，所涉及的医生必须使用 Prescribe（开处方）方法来生成一个电子处方。

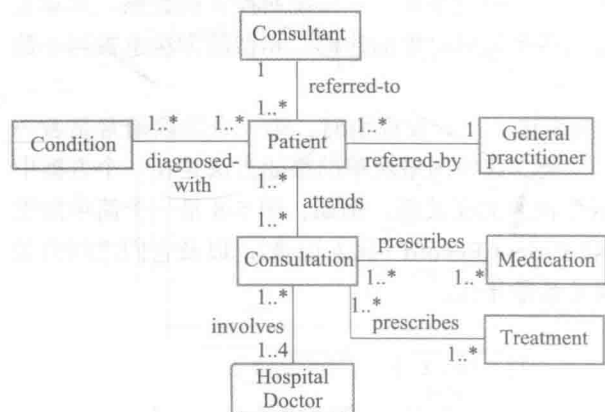


图 5-9 Mentcare 系统中的类和关联关系

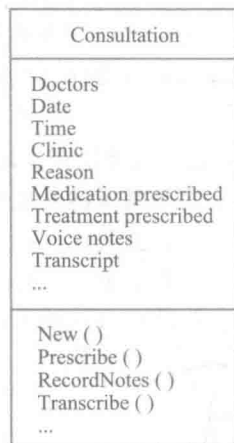


图 5-10 一个 Consultation（诊疗）类

5.3.2 泛化

泛化 (generalization) 是用于管理复杂性的一种日常技术。我们在日常生活中并不是从经历的所有事情的详细特性中进行学习的, 而是学习通用的类 (动物、汽车、房屋等) 以及这些类的特性。在此基础上我们通过对事物进行分类来复用这些知识, 并关注它们与它们所属的类之间的差异。例如, 松鼠和老鼠都是“啮齿类动物”类的成员, 因此共享啮齿类动物的特性。通用的陈述适合所有的类成员, 例如, 所有的啮齿类动物都有咬东西的牙齿。

在建模系统时, 检查一下系统中的类, 看看是否存在泛化和创建类的空间经常很有用。这意味着通用的信息只需要在一个地方保存即可。这是一个好的设计实践, 因为这就意味着如果有变更请求, 那么不需要在系统的所有类里面寻找可能受变更影响的地方。可以在最高的泛化层次上进行变更。在 Java 等面向对象语言中, 泛化使用语言中内置的类继承的机制来实现。

UML 有一个特定类型的关联关系来表示泛化, 如图 5-11 所示。泛化表示为一个向上指向更泛化的类的箭头。这个例子表示全科医师 (General practitioner) 和医院医生 (Hospital doctor) 可以被泛化为医生 (Doctor), 而医院医生包含 3 种类型: 实习医生 (Trainee Doctor), 这种医生刚从医学院毕业, 必须有人指导; 注册医生 (Registered Doctor), 这种医生可以作为顾问医生团队的一员独立工作; 顾问医生 (Consultant), 这种医生是具有完全决策责任的高级医生。

在泛化关系中, 与更高层的类相关的属性和操作同样也与较低层的类相关。较低层的类是从它们的父类那里继承属性和操作的子类。这些较低层类可以增加更加特定的属性和操作。

例如, 所有医生都有姓名和电话号码。所有医院医生都有一个员工编号并携带寻呼机。全科医师没有这些属性, 因为他们是独立工作的, 但是他们有个人诊所名称和地址。图 5-12 显示了这个泛化层次的一部分, 其中扩展了 Doctor 类的类属性。与 Doctor 类相关的操作是让医生注册 Mentcare 系统以及取消注册。

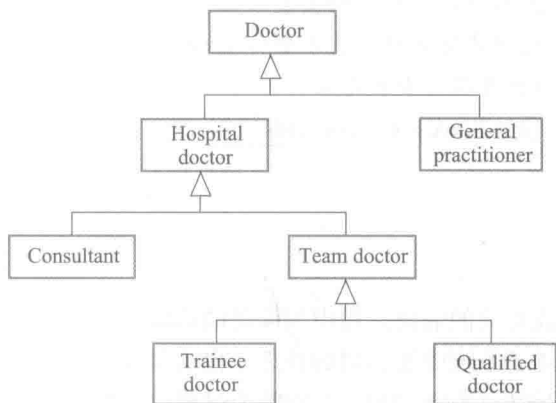


图 5-11 一个泛化层次

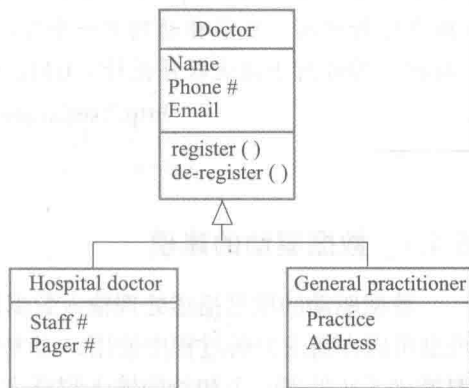


图 5-12 增加了细节的泛化层次

5.3.3 聚集

现实世界中的对象经常由不同的部分组成。例如, 一个课程的学习包可能包括一本书、PowerPoint、课堂测验、推荐阅读。有时候需要在系统模型中描述这种组成关系。UML 为

此提供了一种类与类之间的特殊的关联关系类型，称为聚集（aggregation），其意思是一个对象（整体）由其他对象（部分）组成。为了定义聚集，在链接关系中表示整体的类那一端加上一个菱形。

图 5-13 表明一个病人记录是一个病人以及不确定数量的诊疗的聚集。也就是说，病人记录保存着病人的个人信息以及每一次医生对其的诊疗记录。

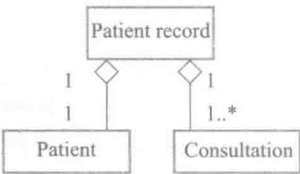


图 5-13 聚集关联

5.4 行为模型

行为模型是关于系统在运行时的动态行为的模型。这种模型描述了当系统对来自环境的激励进行响应时发生什么或者应该发生什么。激励可以是数据或事件。

- 1. 系统必须处理的数据产生了。数据的产生触发了相应的处理过程。
- 2. 一个事件发生了，触发了系统处理。事件可以有相关联的数据，虽然并非总是如此。

许多业务系统是主要由数据驱动的数据处理系统。这些系统由输入给系统的数据来控制，外部事件处理相对较少。它们的处理包括在数据上的一系列动作以及产生一个输出。例如，一个电话账务系统会接收一个客户所发出的呼叫信息，计算这些呼叫的费用并为该客户生成一个账单。

与之相比，实时系统通常都是事件驱动的，数据处理比较少。例如，一个固定电话交换系统通过产生拨号音对“听筒被激活”事件进行响应，通过记录电话号码对听筒事件进行响应。



数据流图

数据流图（Data-flow diagram, DFD）是一种展现功能性视角的系统模型，其中每个转换（transformation）表示单个的功能或过程。数据流图用于描述数据流如何从一系列处理步骤中流过。例如，一个处理步骤可以是过滤客户数据库中的重复记录。数据在每个步骤上进行转换，然后移动到下一个阶段。这些处理步骤或转换表示软件过程或功能，其中数据流图被用于描述软件设计。UML 中的活动图可以用于表达数据流图。

<http://software-engineering-book.com/web/dfd/>

5.4.1 数据驱动的建模

数据驱动模型描述处理输入数据以及生成相关的输出过程中所涉及的动作序列。这种模型可以在需求分析过程中使用，因为它们描述了系统中端到端的处理。也就是说，这种模型描述了从处理一个初始的输入到产生相应的输出（系统的响应）的整个过程中所发生的整个动作序列。

数据驱动模型是最早使用的图形化软件模型之一。20 世纪 70 年代，结构化设计方法将数据流图（DFD）作为一种描述系统中的处理步骤的方法使用。数据流模型很有用，因为追踪并描述与特定过程相关的数据是如何在系统中流动的，可以有助于分析人员和设计者理解在此过程中做了什么。数据流图简单、直观，因此对于利益相关者而言比其他类型的模

型更容易理解。通常都可以向潜在的系统用户解释这些数据流图，从而让他们参与模型的确认。

数据流图可以用 UML 中的活动图（见 5.1 节）来表示。图 5-14 是一个简单的活动图，其中描述了胰岛素泵软件中所包含的处理过程链。从中可以看到表示为活动（圆角矩形）的处理步骤以及表示为对象（矩形）的在步骤间流动的数据。

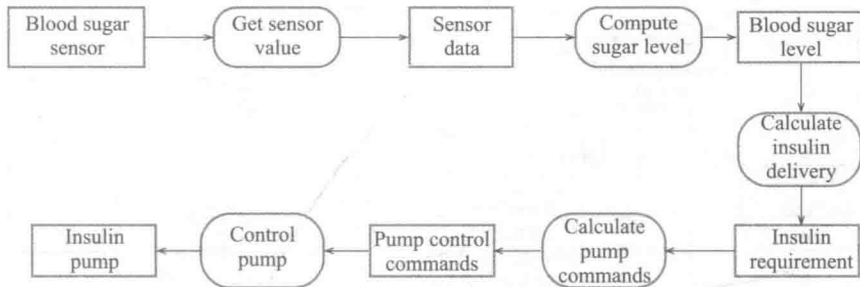


图 5-14 胰岛素泵运行的活动模型

另一种描述系统中的处理序列的方法是使用 UML 顺序图。此前已经展示过如何使用顺序图进行交互建模，但是如果在画顺序图时消息都是从左到右发送，那么顺序图可以描述系统中顺序化的数据处理。图 5-15 描述了这一点，使用一个顺序模型描述订单处理以及发送订单给供应商。顺序模型强调系统中的对象，而数据流图强调操作或活动。在实践中，非专家的普通用户似乎觉得数据流图更直观一些，而工程师则倾向于使用顺序图。

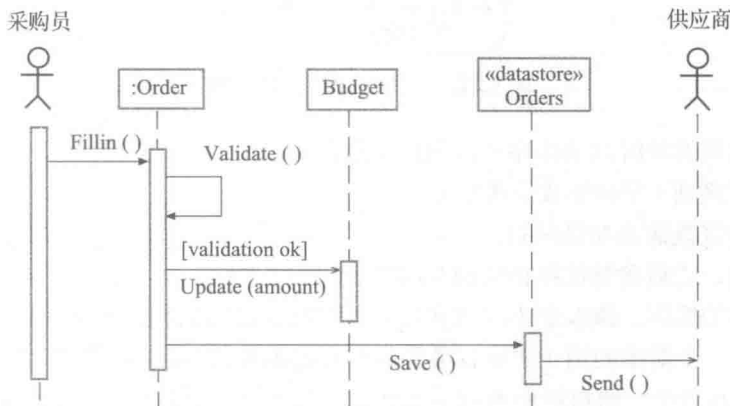


图 5-15 订单处理

5.4.2 事件驱动的建模

事件驱动的建模描述一个系统如何对外部和内部事件做出响应。这种模型建立在系统具有有限数量的状态以及事件（激励）可以导致状态间的转换的假设基础上。例如，一个阀门控制系统可以在收到操作人员命令（激励）时从“阀门开启”状态转移到“阀门关闭”状态。这种系统观点特别适合于实时系统。事件驱动的建模在实时系统的设计和文档化（第 21 章）中得到了广泛使用。

UML 通过状态图（state diagram）支持基于事件的建模，而 UML 状态图是基于此前提出的状态图（state chart）的（Harel 1987）。状态图描述了系统状态以及导致状态间转换的事件。

状态图不会描述系统中的数据流，但是可以包含关于每个状态中所执行的计算的额外信息。

本书使用一个关于简单的微波炉控制软件的例子来介绍事件驱动的建模（见图 5-16）。真正的微波炉比这个系统复杂得多，不过这个简化的系统更容易理解。这个简单的微波炉有：一个选择全功率还是半功率的开关，一个输入加热时间的数字键盘，一个启动 / 停止按钮，一个字母数字显示器。

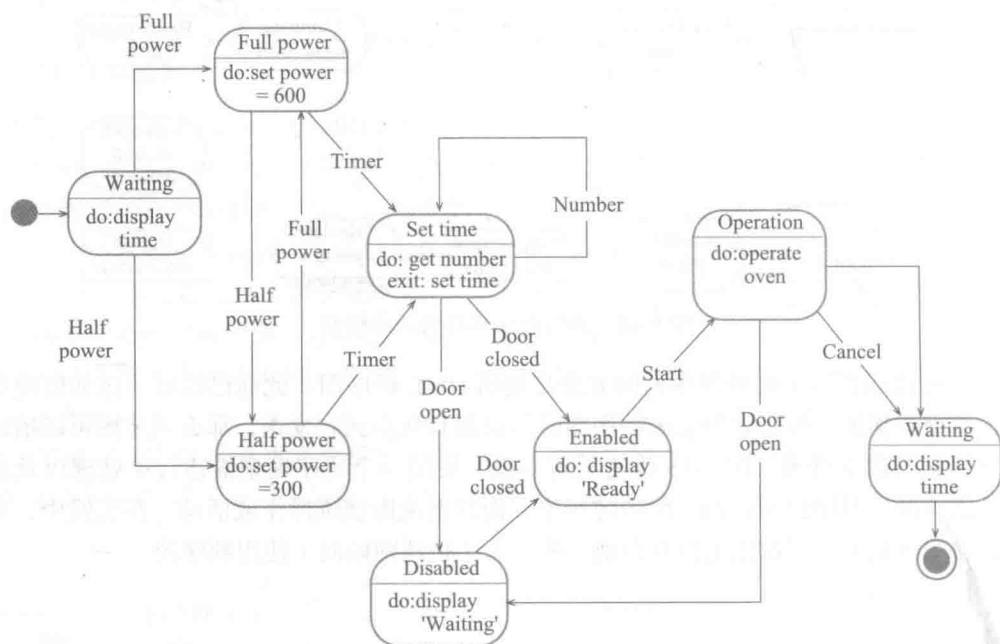


图 5-16 一个微波炉的状态图

这里对使用微波炉时的动作序列做出以下假设。

1. 选择功率级别（半功率或全功率）；
2. 使用数字键盘输入加热时间；
3. 按启动键，之后食物按照指定的时间进行加热。

因为安全性的原因，微波炉不应该在门开着的时候进行加热，并且当加热完成时蜂鸣器响起。微波炉有一个简单的用于显示各种警告和报警消息的显示器。

在 UML 状态图中，圆角矩形表示系统状态。每个状态可以包含对在该状态中所执行的动作的一个简要描述（在“do”之后）。带标签的箭头表示驱动从一个状态转换到另一个状态的激励。可以像在活动图中一样使用实心圆表示开始和终结状态。

从图 5-16 中可以看出，系统开始于一个等待（Waiting）状态，并且首先对全功率（Full power）或半功率（Half power）按钮进行响应。用户可以在选择了其中一个模式后改变主意，再次按下另一个按钮。设置时间（Set time）后，如果门是关着的，那么启动按钮激活（Enabled）。按下启动按钮启动微波炉运转，按照指定时间进行加热。然后加热过程结束，系统返回等待状态。

基于状态的建模的问题是，可能的状态的数量会快速增长。因此，对于大的系统模型，需要在模型中隐藏细节。一种隐藏细节的方式是使用“父状态”（superstate）的概念，在其中封装了一些独立的状态。这种父状态看起来像是高层模型中的单个状态，但是可以在另一个

图中展开以显示更多的细节。为了说明这个概念，考虑图 5-16 中的（运转）状态。这是一个可以展开的父状态，如图 5-17 所示。

Operation 状态包含一些子状态。图中表明运转开始于微波炉状态检查（check status），如果发现任何问题则会发出警报并且不允许运转。加热时会按照所指定的时间运行微波炉的电机（run generator）；完成加热后蜂鸣器响起。如果在运转过程中门被打开，那么系统进入不可用（disabled）状态，如图 5-17 所示。

系统状态模型提供了关于事件处理的概览，但是通常还要在此基础上扩展更详细的关于激励以及系统状态的描述。可以使用一个表格来列出状态以及激励状态转换的事件，同时包括对每个状态和事件的描述。图 5-18 展现了对每个状态以及产生驱动状态转换的激励的表格化描述。

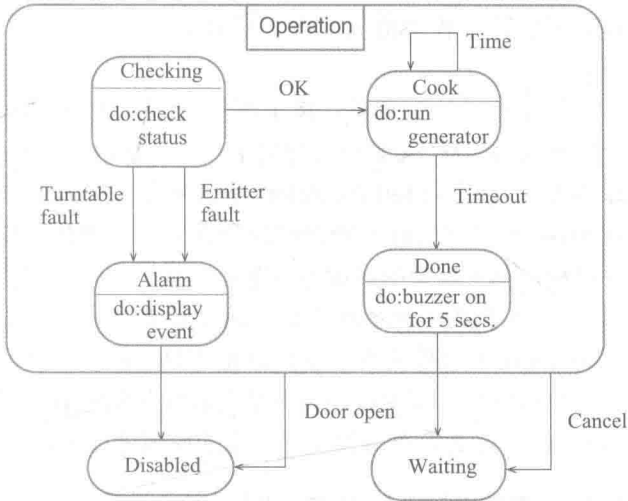


图 5-17 Operation 状态的状态模型

状 态	描 述
等待 (Waiting)	微波炉在等待输入。显示器显示当前时间
半功率 (Half power)	微波炉功率被设置为 300 瓦特。显示器显示“半功率”
全功率 (Full power)	微波炉功率被设置为 600 瓦特。显示器显示“全功率”
设置时间 (Set time)	按照用户的输入值设置加热时间。显示器显示所选择的加热时间并在设置时间时进行更新
未激活 (Disabled)	微波炉运转出于安全原因未激活。微波炉内部的灯亮起。显示器显示“未就绪”
激活 (Enabled)	微波炉运转被激活。微波炉内部的灯关闭。显示器显示“准备加热”
运转 (Operation)	微波炉正在运转。微波炉内部的灯亮起。显示器显示计时器倒计时。加热完成后，蜂鸣器响起 5 秒钟。微波炉灯亮起。显示器在蜂鸣器响的过程中显示“加热完成”
激 励	描 述
半功率 (Half power)	用户按下了半功率按钮
全功率 (Full power)	用户按下了全功率按钮
计时器 (Timer)	用户按下了其中一个计时器按钮
数字 (Number)	用户按下了一个数字键
门打开 (Door open)	微波炉门开关没有关上
门关上 (Door closed)	微波炉门开关关上了
启动 (Start)	用户按下了启动键
取消 (Cancel)	用户按下了取消键

图 5-18 微波炉的状态和激励

5.4.3 模型驱动的工程

模型驱动的工程（Model-Driven Engineering, MDE）是一种软件开发方法，其中开

发过程的主要产出物是模型而不是程序 (Brambilla, Cabot, and Wimmer 2012)。在硬件/软件平台上运行的程序是从模型自动生成的。模型驱动的工程的支持者坚持认为这一点提高了软件工程的抽象层次,从而使工程师不再需要关注编程语言的细节或者执行平台的细节。

模型驱动的工程是从模型驱动的体系结构 (Model-Driven Architecture, MDA) 基础上发展起来的。MDA 是由对象管理组织 (Object Management Group, OMG) 提出的一种新的软件开发范型 (Mellor, Scott, and Weise 2004)。MDA 关注软件开发的设计和实现阶段,而 MDE 则关注软件工程过程的所有方面。因此,基于模型的需求工程、基于模型的开发软件过程、基于模型的测试等话题是 MDE 的一部分,但在 MDA 中没有考虑。

MDA 作为一种系统工程方法已经被一些大公司采用来支持他们的开发过程。本章关注将 MDA 用于软件实现而不是讨论 MDE 中更加全面的方面。更加全面的模型驱动的工程推广得并不顺利,很少有公司在整个软件开发生命周期中采用该方法。den Haan 在他的博客中对 MDE 为什么没有得到广泛采用的原因进行了讨论 (den Haan 2011)。

5.5 模型驱动的体系结构

模型驱动的体系结构 (Mellor, Scott, and Weise 2004; Stahl and Voelter 2006) 是一种关注模型的软件设计和实现方法,使用了 UML 模型的一个子集来描述系统,其中会创建不同抽象层次上的模型。从原则上讲,在没有人干涉的情况下,从一个高层的平台无关模型是可以生成一个可运行程序的。

模型驱动的体系结构 (MDA) 方法建议应当产生以下 3 种类型的抽象系统模型。

1. 计算无关模型 (Computation Independent Model, CIM)。CIM 对系统中使用的重要的领域抽象进行建模,因此有时被称为领域模型。可以开发几种不同的 CIM,反映系统的不同视图。例如,可以有一个信息安全 CIM,明确资产等重要信息安全抽象;以及一个角色和一个病人记录 CIM,描述病人、诊疗等抽象。

2. 平台无关模型 (Platform-Independent Model, PIM)。PIM 在不涉及实现的情况下对系统的运转进行建模。PIM 通常使用 UML 模型描述,显示静态系统结构以及系统如何响应外部和内部事件。

3. 平台相关模型 (Platform-Specific Model, PSM)。PSM 是对平台无关模型转换后得到的,对于每个应用平台都有一个单独的 PSM。原则上讲,PSM 可能存在多个层次,其中每个层次增加一些平台相关的细节。因此,第一个层次的 PSM 可以是中间件相关的但是数据库无关的。当选择了一个特定的数据库之后,就可以生成一个数据库相关的 PSM。

基于模型的工程允许工程师在较高的抽象层次上考虑系统,不用关心它们的实现细节。这减少了错误的可能性,加快了设计和实现过程,同时还可以创建可复用、平台无关的应用模型。通过使用强大的工具,可以利用同样的模型为不同的平台生成系统实现。因此,为了让系统适应一些新的平台技术,可以为该平台编写一个模型转换器。有了这种转换器,所有平台无关模型就可以快速在新平台上落地了。

MDA 是建立在模型间的转换可以通过软件工具进行定义和自动化执行的思想基础上的,如图 5-19 所示。该图还描述了自动转换的最终层次,其中对 PSM 进行了转换以生成可以在指定的软件平台上运行的代码。因此,至少在原则上,可执行软件可以从一个高层系统模型生成。

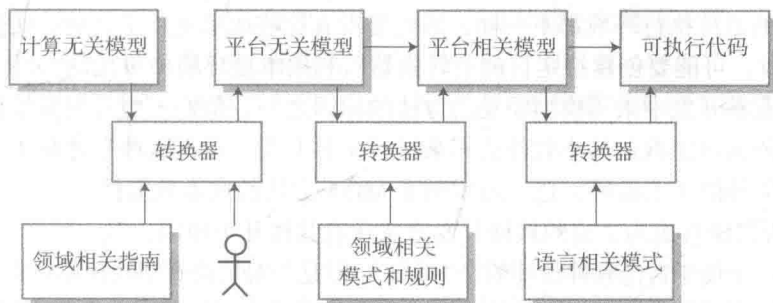


图 5-19 MDA 转换

在实践中，很少能做到完全自动化地从模型转换到代码。从高层的 CIM 到 PIM 的转换仍然是一个研究问题。对于生产系统，人为干预（如图 5-19 中的线条人形所示）通常都必不可少。自动化的模型转换的一个特别困难的问题是，需要将在不同的 CIM 中使用的概念链接起来。例如，一个信息安全 CIM 中包含角色驱动访问控制的角色的概念，可能要映射到医院 CIM 中的工作人员的概念上。只有同时理解信息安全和医院环境的人才可以做出这个映射。

平台无关模型向平台相关模型的转换是一个较简单的技术问题。有一些商业化工具和开源工具（Koegel 2012）可以使用，这些工具提供了从 PIM 到 Java 和 J2EE 等通用平台的转换器。这些工具使用了一个广泛的平台相关的规则和模式库来实现 PIM 到 PSM 的转换。系统中的每个 PIM 都可能有多多个相应的 PSM。如果一个软件系统要在不同的平台（例如 J2EE 和 .NET）上运行，那么在原则上只需要维护一个 PIM。针对每个平台的 PSM 可以自动生成（见图 5-20）。

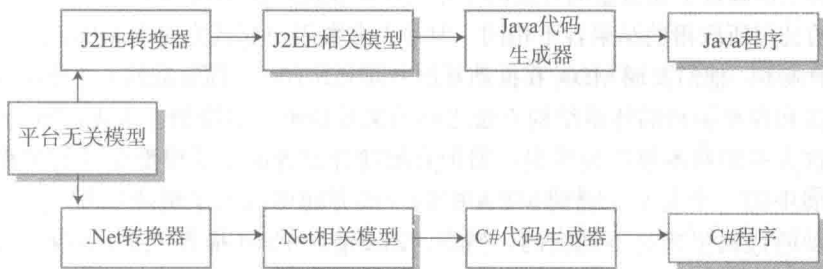


图 5-20 多个平台相关模型

虽然 MDA 支持工具包括平台相关的转换器，这些工具有时候只能部分支持 PIM 到 PSM 的转换。系统的执行环境不仅仅是 J2EE 或者 Java 等标准的执行平台，还可能包括为一个企业创建的其他应用系统和特定应用库、外部服务以及用户界面库。



可执行的 UML

模型驱动的工程背后的基本思想是，完全自动化地从模型转换到代码应该是可能的。为了达到这个目标，必须能够构造可以被编译为可执行代码、带有清晰定义的含义的图形化模型。还需要一种方法，向图形化模型增加关于模型中所定义操作的实现方式的信息。使用 UML 2 的一个称为可执行的 UML 或 xUML 的子集可以实现这一点（Mellor and Balcer 2002）。

<http://software-engineering-book.com/web/xuml/>

每个公司的系统执行环境都不一样，因此是没有这些成品支持工具的。因此，当一个组织引入 MDA 时，可能要创建特定目的的转换器以利用本地环境中可用的条件。这是为什么许多公司不愿意在开发中采用模型驱动的方法的原因之一。他们不想开发或维护他们自己的工具，更不想依赖可能歇业的小软件公司来进行工具开发。没有这些专业化工具，基于模型的开发就要求额外的手工编码，这一点削弱了 MDA 方法的成本效益性。

MDA 之所以没有成为主流的软件开发方法还有其他几个原因。

1. 模型是一个便于讨论软件设计的好的方法。但是，对讨论有用的抽象对实现而言并不总是正确的抽象。你可能会出于复用成品应用系统的考虑而决定使用一个完全不同的实现方法。

2. 对于大多数复杂系统，实现不是主要的问题——需求工程、信息安全和可依赖性、遗留系统集成和测试都是更重要的问题。因此，使用 MDA 的收益就很有有限了。

3. 追求平台无关性只对大型、长生命周期的系统有意义，其中平台会在系统生命周期过程中逐渐变得过时。对于针对标准平台开发的软件产品和信息系统（例如 Windows 和 Linux），引入 MDA 方法以及准备相关工具的成本可能会超出使用 MDA 获得的收益。

4. 在 MDA 方法发展过程的同一时期，被广泛采用的敏捷方法成功地将开发人员的注意力从模型驱动上吸引走了。

MDA 的成功故事（OMG 2012）大多数都来自于开发系统产品的公司，包括硬件和软件。这些产品中的软件有很长的生命周期，在此过程中可能要进行修改以反映硬件技术的变化。应用领域（汽车、空中交通管制等）经常是已经得到了充分理解的，因此可以被形式化表示为一个 CIM。

Hutchinson 及其同事（Hutchinson, Rouncefield, and Whittle 2012）报告了 MDA 在工业界的使用，他们的工作确认了模型驱动开发的使用已经在系统产品中取得了成功。他们的评估表明采用该方法的公司所取得的结果各不相同，但是大多数用户都认为使用 MDA 提高了生产率并且降低了维护成本。他们发现 MDA 在推动复用方面尤其有用，而这造就了主要的生产率提升。

敏捷方法和模型驱动的体系结构方法之间的关系还不是很清楚。事先进行全面的建模的思想与敏捷宣言中的基本思想相冲突，很少有敏捷开发者能接受模型驱动的工程。Ambler，敏捷方法发展中的一个先驱，提到 MDA 中的一些方面可以用于敏捷过程（Ambler 2004），但是认为自动的代码生成是不现实的。然而，Zhang 和 Patel 报告了摩托罗拉在使用敏捷方法和自动化的代码生成方面的成功经验（Zhang and Patel 2011）。

要点

- 一个模型是一个系统的抽象视图，其中有意忽略了一些系统细节。可以开发多个互补的系统模型来表示系统的上下文、交互、结构和行为。
- 上下文模型描述一个所建模的系统是如何被置于包含其他系统和过程的环境之中。它们可以帮助定义待开发系统的边界。
- 用况图和顺序图用于描述所设计的系统中的用户与系统之间的交互。用况描述了一个系统与外部参与者之间的交互；顺序图通过显示系统对象之间的交互向其中增加了更多的信息。
- 结构模型显示一个系统的组织和体系结构。类图被用于定义一个系统中的类的静态结构以及它们的关联关系。
- 行为模型用于描述一个执行中的系统的动态行为。这种行为可以从系统所处理的数据的角度或者从激励系统响应的事件的角度进行建模。

- 活动图可以用于建模数据的处理，其中每个活动表示一个处理步骤。
- 状态图用于建模系统在响应内部或外部事件过程中的行为。
- 模型驱动的工程是一种软件开发方法，其中一个系统被表示为一组可以被自动转换为可执行代码的模型。

阅读推荐

任何关于 UML 的介绍性书籍都提供了关于本章所介绍的概念的更多信息。UML 在过去几年中只有一些细微的变化，因此，虽然这些书中的一些内容几乎是 10 年前出版的，但仍然不过时。

《Using UML: Software Engineering with Objects and Components, 2nd ed》是一本关于 UML 在系统规格说明和设计中的使用的简短、易读的介绍。这本书对于学习和理解 UML 相关的概念是很好的，虽然与 UML 参考手册中完整的 UML 描述相比，没有那么全面。(P. Stevens with R. Pooley, Addison-Wesley, 2006)

《Model-driven Software Engineering in Practice》是一本全面介绍模型驱动的方法的书籍，关注模型驱动的设计和实现。除了 UML，该书还介绍了领域相关的建模语言的发展。(M. Brambilla, J. Cabot, and M. Wimmer. Morgan Claypool, 2012)

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap5/>

支持视频的链接: <http://software-engineering-book.com/videos/requirements-and-design/>

练习

- 5.1 解释建模所开发的系统的上下文为什么很重要。如果软件工程师不理解系统上下文可能出现一些错误，请举出两个例子。
- 5.2 如何使用一个已经存在的系统模型？解释为什么这样一个系统模型并不一定是完整和正确的。如果你在开发一个新系统的模型情况还是这样吗？
- 5.3 你将参与开发一个用来规划大规模活动和聚会（例如婚礼、毕业典礼和生日聚会）的系统。使用一个活动图对这个系统进行过程上下文的建模，显示规划一个聚会所涉及的活动（例如预订场地、安排邀请等）以及每个阶段会用到的系统元素。
- 5.4 针对 Mentcare 系统，提出一组用况来描述一个给病人看病并开药方和治疗处方的医生和 Mentcare 系统之间的交互。
- 5.5 开发一个顺序图来描述一所大学中的一个学生注册一门课程时所涉及的交互。课程可能有人数限制，因此注册过程必须包含针对是否还有空位的检查。假设学生通过访问一个电子课程目录来找出可选的课程。
- 5.6 仔细观察你所使用的电子邮件系统中消息和邮箱是如何表示的。建模为了表示邮箱和电子邮件消息而需要在系统实现中使用的对象类。
- 5.7 基于你对于银行 ATM 机的经验，画一个活动图来建模当一个客户从机器中提取现金时所涉及的数据处理。
- 5.8 为同一个系统画一个顺序图。解释你在建模系统行为时为什么要同时开发活动图和顺序图。
- 5.9 针对以下控制软件开发状态图：

- 针对不同类型的衣物有不同洗涤程序的自动洗衣机；
- 一个 DVD 播放器的软件；
- 手机上的摄像头的控制软件（忽略闪光灯）。

5.10 你是一个软件工程经理，你的团队中的一个资深成员提出应当使用模型驱动的工程来开发一个新系统。你在决定是否应该将该方法引入软件开发中时要考虑哪些因素？

参考文献

- Ambler, S. W. 2004. *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd ed. Cambridge, UK: Cambridge University Press.
- Ambler, S. W., and R. Jeffries. 2002. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.
- Booch, G., J. Rumbaugh, and I. Jacobson. 2005. *The Unified Modeling Language User Guide*, 2nd ed. Boston: Addison-Wesley.
- Brambilla, M., J. Cabot, and M. Wimmer. 2012. *Model-Driven Software Engineering in Practice*. San Rafael, CA: Morgan Claypool.
- Den Haan, J. 2011. "Why There Is No Future for Model Driven Development." <http://www.theenterprisearchitected.eu/archive/2011/01/25/why-there-is-no-future-for-model-driven-development/>
- Erickson, J., and K. Siau. 2007. "Theoretical and Practical Complexity of Modeling Methods." *Comm. ACM* 50 (8): 46–51. doi:10.1145/1278201.1278205.
- Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.
- Hull, R., and R. King. 1987. "Semantic Database Modeling: Survey, Applications and Research Issues." *ACM Computing Surveys* 19 (3): 201–260. doi:10.1145/45072.45073.
- Hutchinson, J., M. Rouncefield, and J. Whittle. 2012. "Model-Driven Engineering Practices in Industry." In *34th Int. Conf. on Software Engineering*, 633–642. doi:10.1145/1985793.1985882.
- Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.
- Koegel, M. 2012. "EMF Tutorial: What Every Eclipse Developer Should Know about EMF." <http://eclipsesource.com/blogs/tutorials/emf-tutorial/>
- Mellor, S. J., and M. J. Balcer. 2002. *Executable UML*. Boston: Addison-Wesley.
- Mellor, S. J., K. Scott, and D. Weise. 2004. *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley.
- OMG. 2012. "Model-Driven Architecture: Success Stories." http://www.omg.org/mda/products_success.htm
- Rumbaugh, J., I. Jacobson, and G. Booch. 2004. *The Unified Modelling Language Reference Manual*, 2nd ed. Boston: Addison-Wesley.
- Stahl, T., and M. Voelter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.
- Zhang, Y., and S. Patel. 2011. "Agile Model-Driven Development in Practice." *IEEE Software* 28 (2): 84–91. doi:10.1109/MS.2010.85.

体系结构设计

目标

本章的目标是介绍软件体系结构和体系结构设计概念。阅读完本章后，你将：

- 理解为什么软件的体系结构设计很重要；
- 理解在体系结构设计过程中必须做出的关于软件体系结构的决策；
- 了解体系结构模式的思想，这是一种经过实践验证、可以在系统设计中复用的软件体系结构的组织方式；
- 理解为什么应用特定的体系结构模式可以被用于事务处理和语言处理系统。

体系结构设计关注理解一个软件系统应当如何组织，以及设计该系统的整体结构。在第2章所介绍的软件开发过程中，体系结构设计是软件设计过程的第一个阶段。体系结构设计是设计和需求工程之间的关键性衔接环节，因为它会确定组成一个系统的主要结构构件以及它们之间的关系。体系结构设计过程的输出是一个体系结构模型，该模型描述系统如何被组织为一组相互通信的构件。

在敏捷过程中，得到广泛认同的一点是，一个敏捷开发过程的早期阶段应该关注设计一个整体的系统体系结构。体系结构的增量开发通常都不会成功。根据变化重构构件通常相对容易。然而，重构系统体系结构却很昂贵，因为可能需要修改大部分系统构件以使它们能够适应体系结构的变化。

为了帮助读者理解系统体系结构是什么意思，可以参考图6-1。图6-1显示了一个打包机器人系统的体系结构的抽象模型。这个机器人系统可以对不同类型的物品进行打包。它使用一个视觉构件来挑选传送带上的物品，识别物品类型，并且选择正确的打包方式。接着，该系统将物品从传送带上取下来进行打包，然后将打好包的物品放在另一个传送带上。这个体系结构模型显示了这些构件以及它们之间的连接关系。

在实践中，需求工程过程和体系结构设计过程之间存在显著的重叠。理想情况下，系统规格说明不应当包含任何设计信息。然而，这个设想是不现实的，除非面向的是非常小的系统。需要识别出主要的体系结构构件，因为这些构件反映了系统的高层特征。因此，作为需求工程过程的一部分，要提出一个

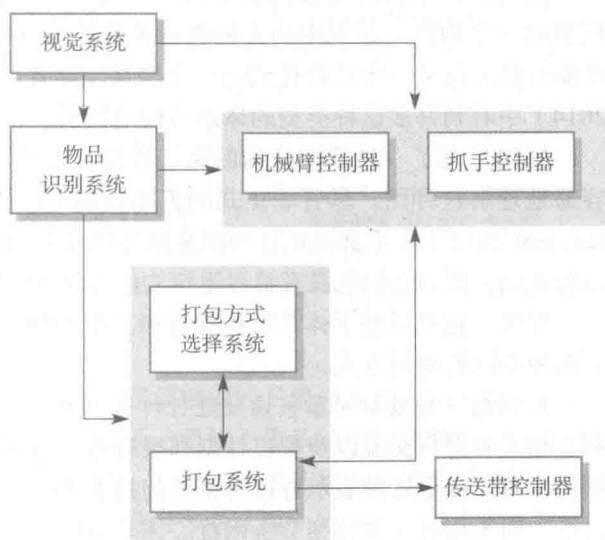


图 6-1 打包机器人控制系统的体系结构

抽象的系统体系结构，其中将一组组的系统功能或特征与大规模的构件或子系统关联起来。接下来，要使用这一分解结构来与利益相关者讨论需求以及更加详细的系统特征。

可以在两个抽象层次上设计软件体系结构，本书将它们分别称为小体系结构和大体系结构。

1. 小体系结构关注单个程序的体系结构。在这个层次上，我们关注单个的程序是如何被分解为构件的。本章主要关注程序体系结构。

2. 大体系结构关注包括其他系统、程序和程序构件的复杂企业系统体系结构。这些企业系统可以分布在不同的计算机上，这些计算机可以由不同的公司拥有和管理。第 17 和 18 章将会介绍这种大体系结构。

软件体系结构很重要，因为它会影响一个系统的性能、健壮性、分布性和可维护性 (Bosch 2000)。正如 Bosch 所说，单个构件实现了功能性的系统需求，但是对于非功能性系统特性的主导影响因素是系统的体系结构。Chen 等人 (Chen, Ali Babar, and Nuseibeh 2013) 在一个名为“对体系结构有显著影响的需求”的研究中确认了这一点，他们发现非功能性需求对于系统体系结构的影响最大。

Bass 等人 (Bass, Clements, and Kazman 2012) 认为明确地设计并描述软件体系结构有以下 3 个好处。

1. 利益相关者交流。体系结构是一种可以作为许多不同利益相关者讨论的焦点的系统的表示。

2. 系统分析。在系统开发的早期阶段明确系统的体系结构要求进行一些分析。体系结构设计决策对于系统是否能够满足性能、可靠性、可维护性等关键需求具有深远的影响。

3. 大范围复用。体系结构模型是关于系统如何组织以及构件如何互操作的一个紧凑、可管理的描述。具有相似需求的系统经常具有相同的系统体系结构，因此可以支持大范围的软件复用。

如第 15 章所述，产品线体系结构是一种复用方法，其中同一个体系结构在一系列相关的系统中得到复用。

系统体系结构经常使用简单的框图进行非正式的建模，如图 6-1 所示。图中每一个方框表示一个构件。方框中的方框表示该构件被分解为子构件。箭头表示数据和控制信号按照箭头的方向从一个构件传到另一个构件。读者可以在 Booch 的软件体系结构手册 (Booch 2014) 中看到许多这种类型的体系结构的例子。

框图呈现了一种系统结构的高层样貌，来自不同领域参与系统开发过程的人都可以很容易地理解这种图。尽管非正式的框图得到了广泛使用，Bass 等人 (Bass, Clements, and Kazman 2012) 却不喜欢用这种图来描述体系结构。他们认为这些非正式的图是糟糕的体系结构表示，因为它们既没有显示系统构件之间关系的类型也没有显示构件的外部可见属性。

显然，这里反映了体系结构理论和工业实践之间的矛盾，因为程序的体系结构模型有如下两种不同的使用方式。

1. 作为一种鼓励对系统设计进行讨论的方式。一个系统的高层体系结构视图对于与系统利益相关者进行交流以及项目计划都很有用，因为它没有被过多的细节弄得难以理解。利益相关者可以接受这种表示并理解系统的抽象视图。接下来他们可以将系统作为一个整体进行讨论，而不用被过多的细节所困扰。体系结构模型识别出了要开发的关键构件，从而使管理者可以开始分配人员以规划这些系统的开发。

2. 作为一种文档化已经设计好的体系结构的方式。这里的目的是产生一个显示系统中不同的构件、它们的接口以及连接关系的完整的系统模型。支持这种模型的观点认为, 这样一种详细的体系结构描述使得对系统的理解和演化可以更容易。

框图是一种很好的支持参与软件设计过程的人之间进行交流的方式。框图很直观, 领域专家和软件工程师都可以理解并参与关于系统的讨论。管理人员发现框图对于项目计划很有帮助。对于很多项目, 框图是唯一的体系结构描述。

理想情况下, 如果要详细描述一个系统的体系结构, 最好能使用一种更严格的软件体系结构描述的表达法。为此已经提出了一系列不同的体系结构描述语言 (Bass, Clements, and Kazman 2012)。一个更加详细和完整的描述意味着对体系结构构件之间关系误解的可能性会降低。然而, 开发一个详细的体系结构描述是昂贵而耗时的。在实践中无法得知这样做在成本效益上是否合算, 因此这种方法没有得到广泛使用。

6.1 体系结构设计决策

体系结构设计是一个创造性的过程, 在这个过程中你会设计一个满足系统功能性和非功能性需求的系统组织结构。并不存在公式化的体系结构设计过程。具体的体系结构设计过程取决于所开发的系统的类型、系统架构师的背景和经验, 以及系统的特定需求。因此, 最好是将体系结构设计作为一系列要做出的决策而不是一个活动序列来考虑。

在体系结构设计过程中, 系统架构师必须做出一系列深刻影响系统及其开发过程的结构决策。基于他们的知识和经验, 他们必须考虑图 6-2 中所显示的这些基本问题。

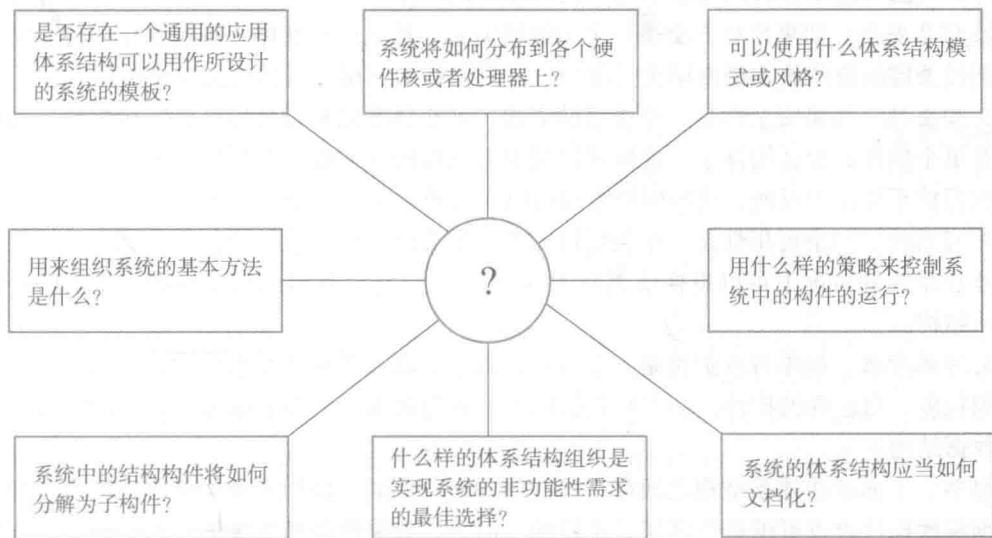


图 6-2 体系结构设计决策

虽然每个软件系统都是独特的, 但是同一个应用领域中的系统经常具有相似的、反映领域的基本概念的体系结构。例如, 应用产品线是基于一个核心体系结构构建的应用, 这种核心体系结构具有满足特定客户需求的变体。当设计一个系统体系结构时, 你必须确定你的系统和更广阔的应用类型有什么共性, 确定来自这些应用体系结构中的知识有多少可以复用。

对于嵌入式系统以及针对个人电脑和移动设备设计的应用, 不需要为系统设计一个分布

式体系结构。然而,大多数大型系统是分布式系统,其中系统软件分布在许多不同的计算机上。分布式体系结构的选择是一个影响系统性能和可靠性的关键决策。这一点将在第 17 章中重点介绍。

一个软件系统的体系结构可以基于特定的体系结构模式或风格(这些术语现在的意思差不多)。一个体系结构模式是一种系统组织方式的描述(Garlan and Shaw 1993),例如,客户端/服务器组织或者层次化体系结构。体系结构模式捕捉了一个在不同的软件系统中使用的体系结构的本质特性。在进行系统体系结构决策时应当了解通用的模式、这些模式的适用场合、它们的优势和弱点。6.3 节介绍了几个经常使用的模式。

Garlan 和 Shaw 关于体系结构风格的思想覆盖了图 6-2 中所显示的体系结构基本问题列表中的问题 4~问题 6。必须选择可以满足系统需求的最合适的结构,例如,客户端/服务器或者层次结构。为了分解结构化系统单元,要确定一种策略用于将构件分解为子构件。最终,在控制建模过程中,开发一个系统不同部分之间的总的控制关系模型,并做出关于构件执行如何控制的决策。

由于非功能性系统特性和软件体系结构的密切关系,体系结构风格和结构的选择应当根据系统的非功能性需求来决定。

1. 性能。如果性能是一个关键性需求,那么体系结构设计应当将关键性操作局部化到少量的构件中,尽量让这些构件部署在同一台计算机上而不要分布在网络上。这可能意味着使用一些相对较大的构件而不是小的更细粒度的构件。使用大构件减少了构件间的通信量,因为相关的系统特征之间的大多数交互都发生在一个构件之内。还可以考虑在运行时系统组织中允许系统的多副本部署以及在不同的处理器上执行。

2. 信息安全。如果信息安全是一个关键性需求,那么应该使用一种层次化的体系结构组织,把最关键的资产放在最内层进行保护,并对这些层应用高级的信息安全确认。

3. 安全性。如果安全性是一个关键性需求,那么体系结构设计应该让安全性相关的操作集中在单个构件或少量构件中。这样可以减少安全性确认的成本和问题,并且可以使得提供相关的保护系统成为可能,这些保护系统可以在失效发生时安全地关闭系统。

4. 可用性。如果可用性是一个关键性需求,那么体系结构设计应该包含冗余的构件以便在不停止系统的情况下可以更换或更新构件。第 11 章将会介绍面向高可用性系统的容错系统体系结构。

5. 可维护性。如果可维护性是一个关键性需求,那么系统体系结构设计应当使用容易改变的细粒度、自包含的构件。应当将数据的生产者与数据的消费者相分离,同时应当避免共享的数据结构。

显然,上面这些体系结构之间存在潜在的冲突。例如,使用大型构件改进性能,与使用小的细粒度构件改进可维护性之间是矛盾的。但是,如果性能和可维护性都是重要的系统需求,那么必须要进行权衡折中。有时候可以通过在系统的不同部分使用不同的体系结构模式或风格来实现权衡。信息安全现在几乎总是一个关键性需求,必须设计一个能够保证信息安全同时也能满足其他非功能性需求的体系结构。

评价一个体系结构设计很难,因为真正的体系结构评判标准是,系统在使用时能够在多大程度上满足它的功能性和非功能性需求。然而,可以通过将设计与参考体系结构或通用体系结构模式相比较来进行评价。Bosch 关于一些体系结构模式的非功能性特性的描述(Bosch 2000)可以帮助进行体系结构评价。

6.2 体系结构视图

本章的引言中提到一个软件系统的体系结构模型可以使与软件需求或设计相关的讨论更加聚焦。此外，体系结构模型也可以用于设计的文档化，这样，体系结构可以作为更加详细的系统设计以及实现的基础。这一部分将讨论以下两个与二者都相关的问题：

1. 在设计和描述一个系统的体系结构时，哪些视图或视角是有用的？
2. 应当使用哪些表示法来描述体系结构模型？

在单个图中表示出与一个系统的体系结构相关的所有信息是不可能的，因为一个图形化模型只能显示一个系统视图或视角。它可以显示系统如何分解为模块、运行时进程如何交互，或者系统构件在一个网络上的不同分布方式。因为所有这些视图在不同时候对设计和文档化都是有用的，你通常都需要从多个不同的视图来呈现软件体系结构。

关于需要哪些体系结构视图有不同的观点。Krutchen (Krutchen 1995) 在他的著名的“4+1”软件体系结构视图模型中提出应当有4个基本的体系结构视图，这些视图可以通过共同的用例或场景连接在一起（见图6-3）。他建议了如下这几个视图。

1. 逻辑视图。这个视图将系统中的关键抽象显示为对象或对象类。应该可以将系统需求与这个逻辑视图中的实体联系起来。

2. 进程视图。这个视图显示系统在运行时如何通过相互交互的进程来构成。该视图对于做出关于非功能性系统特性（例如性能、可用性）的判断很有用。

3. 开发视图。这个视图显示软件如何面向开发任务进行分解；也就是说，该视图显示了软件如何分解为由单个开发者或开发团队实现的构件。该视图对于软件开发管理者和程序员都很有用。

4. 物理视图。这个视图显示系统硬件以及软件构件如何分布在系统中的处理器上。该视图对于规划系统部署方案的系统工程师很有用。

Hofmeister 等人 (Hofmeister, Nord, and Soni 2000) 提出使用相似的视图，但向其中增加了一个概念视图。这个视图是一种系统的抽象视图，可以作为将高层需求分解为更详细的规格说明的基础，帮助工程师确定可以复用的构件，以及表示一个产品线（在第15章中介绍）而不是单个系统。图6-1描述了一个打包机器人的体系结构，可以作为一个概念系统视图的例子。

在实践中，一个系统体系结构的概念视图几乎总是在设计过程中开发。这些概念视图用于向利益相关者解释系统体系结构以及告知体系结构设计决策。在设计过程中，其他一些视图也可以在讨论系统的不同方面的时候进行开发，但是一般都不需要从所有的视角出发开发一个完整的描述。将体系结构模式（在下一节中介绍）与一个系统的不同视图相关联也是可能的。

关于软件架构师是否应该使用UML来描述和文档化软件体系结构存在不同的观点。2006年的一个调研 (Lange, Chaudron, and Muskens 2006) 表明，当使用UML时，通常都是以一种非正式的方式进行应用的。该论文的作者认为这不是一件好事。

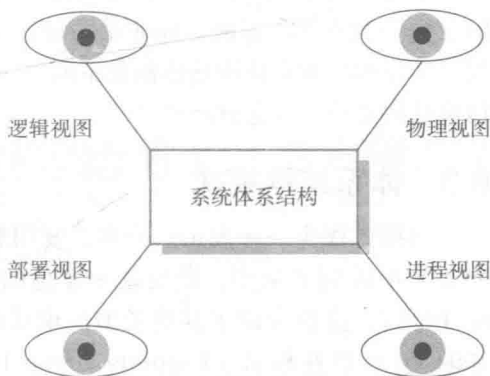


图 6-3 体系结构视图

这一观点不一定正确。UML 被设计用于描述面向对象系统，而在体系结构设计阶段经常要在一个更高的抽象层次上描述系统。对象类过于接近实现，因此对于体系结构描述用处不大。UML 在设计过程中不一定很有用，可以更快编写、更快在白板上画出来的非正式的表达法可能更好。UML 在详细地文档化一个体系结构或者使用模型驱动的开发（如第 5 章中的介绍）时最有价值。

一些研究者（Bass, Clements, and Kazman 2012）提出，使用更加专业化的体系结构描述语言（Architectural Description Language, ADL）来描述系统体系结构。体系结构描述语言的基本元素是构件和连接器，并且包含面向结构良好的体系结构的规则和指南。然而，由于体系结构描述语言是专家语言，领域和应用专家感觉很难理解和使用体系结构描述语言。作为模型驱动开发的一部分使用特定领域的体系结构描述语言可能有一些价值，但它们很可能不会成为主流软件工程实践的一部分。非正式的模型和表示法（例如 UML）依旧是最常使用的系统体系结构描述方式。

敏捷方法的使用者认为详细的设计文档通常都是没用的。因此，开发这种文档是浪费时间。这一观点基本正确。除非是关键性系统，否则从 Krutchen 的 4 个体系结构视图出发开发一个详细的体系结构描述都是不值得的。应该开发的是对于交流有用的视图，而不要担心体系结构文档是否完整。

6.3 体系结构模式

将模式作为一种表示、分享、复用软件系统相关知识的思想已经在软件工程中的一系列领域中得到了采用。激发这一思想的是一本关于面向对象设计模式的书籍（Gamma et al. 1995）。这也带动了其他类型的模式的发展，例如组织设计模式（Coplien and Harrison 2004）、可用性模式（Usability Group 1998）、协作性交互模式（Martin and Sommerville 2004）以及配置管理模式（Berczuk and Appleton 2002）。

体系结构模式是在 20 世纪 90 年代提出的，最初被称为“体系结构风格”（Shaw and Garlan 1996）。一个非常详细厚达五卷的关于面向模式的软件体系结构的系列手册在 1996—2007 年出版（Buschmann et al. 1996; Schmidt et al. 2000; Buschmann, Henney, and Schmidt 2007a, 2007b; Kircher and Jain 2004）。

这一部分将介绍体系结构模式并简要描述一些广泛使用的体系结构模式。模式可以使用标准的方式进行描述（图 6-4 和图 6-5），其中混合使用了叙述性的描述和图。如果想获得关于模式及其使用的更多详细信息，请参考已出版的模式手册。

名 称	MVC (模型 - 视图 - 控制器)
描述	将呈现和交互从系统数据中分离出来。系统被组织为 3 个相互交互的逻辑构件。模型 (Model) 构件管理系统数据以及相关的对这些数据的操作。视图 (View) 构件定义并管理数据呈现给用户的方式。控制器 (Controller) 构件管理用户界面（例如按键、鼠标点击等）并将这些交互传递给视图和模型。见图 6-5
例子	图 6-6 显示了一个使用 MVC 模式进行组织的基于 Web 的应用系统的体系结构
何时使用	当存在多种查看数据以及与数据交互的方式时使用。也可以在未来关于数据的交互和呈现的需求未知时使用
优点	允许数据独立于它的呈现方式进行变更，反之亦然。支持以不同的方式呈现同样的数据，在某一个呈现方式中进行的修改可以在所有呈现方式中显示
缺点	当数据模型和交互比较简单时，可能包含额外的代码以及代码复杂性

图 6-4 模型 - 视图 - 控制器 (MVC) 模式

可以将体系结构模式理解为对好的实践的一种风格化、抽象化的描述，这些实践已经在不同的系统和环境中进行了尝试和测试。因此，一个体系结构模式应当描述一种已经在此前的系统中得到成功应用的系统的组织。它应当包含何时适合以及何时不适合使用该模式，还应包含关于该模式的优缺点的详细描述等信息。

图 6-4 描述了广为人知的模型 - 视图 - 控制器模式。该模式是许多基于 Web 的系统中交互管理的基础，并且得到了大多数语言框架的支持。这种风格化的模式描述包括一个模式名称、一段简要描述、一个图形化模型，以及一个使用了这种模式的系统的例子。其中还应包含关于该模式应该在何种情况下使用以及模式的优缺点的信息。

与 MVC 模式相关的体系结构的图形化模型显示在图 6-5 和图 6-6 中。这两幅图从不同的视图呈现了体系结构：图 6-5 是一个概念视图，图 6-6 显示了当该模式用于基于 Web 的系统中的交互管理时，一个运行时系统的体系结构。

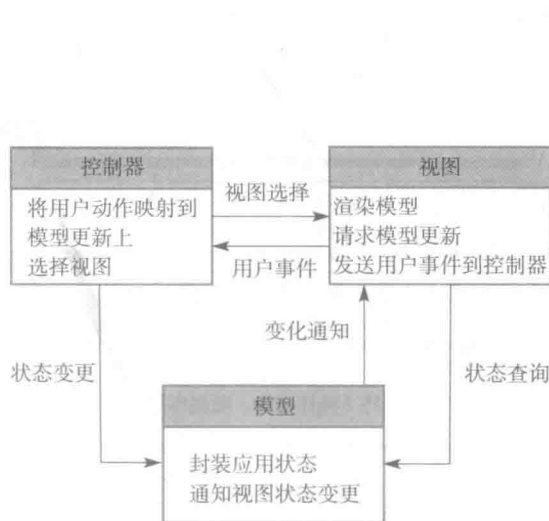


图 6-5 模型 - 视图 - 控制器的组织

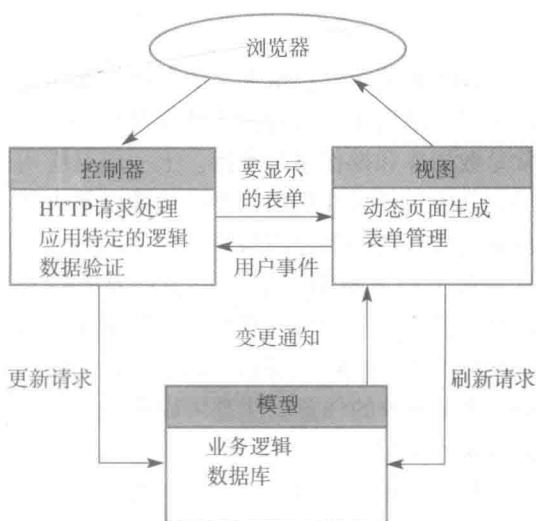


图 6-6 使用 MVC 模式的 Web 应用体系结构

由于篇幅原因，不可能介绍可以在软件开发中使用的所有的通用模式。因此，这里选择了一些广泛使用且反映了好的体系结构设计原则的模式进行介绍。

6.3.1 分层体系结构

分离和独立性的思想是体系结构设计的基础，因为这可以使变更被局部化。图 6-4 中所示的 MVC 模式将系统的元素分离，允许它们独立变更。例如，增加一个新的视图或者改变一个现有的视图可以在不改变所依赖的模型中的数据的情况下实现。分层体系结构模式是另一种实现分离和独立性的方式。这一模式在图 6-7 中进行了描述，系统功能被组织为多个分离的层次，每个层次只依赖于紧邻的下一层所提供的设施和服务。

这一分层的方法支持系统的增量开发。一个层次开发好之后，该层次所提供的一些服务可以提供给用户。该体系结构也是可变化和可前移的。如果接口不变，那么一个功能进行了扩展的新层可以在不影响系统的其他部分的情况下取代一个已有的层。而且，当一个层次的接口发生变化或者增加了新的设施时，只有相邻的层次受到影响。由于分层的系统将机器的依赖局部化了，这使提供一个应用系统的多平台实现变得更容易了。只需将与机器有依赖关

系的层重新实现，便可以适应不同的操作系统或数据库的设施。

名 称	分层体系结构
描述	将系统组织为多个层次，每个层次与一些相关的功能相联系。每个层次向其上的一层提供服务，因此那些最底层次表示很有可能在整个系统中使用的核心服务（见图 6-8）
例子	一个数字化学习系统的分层模型支持学校中所有科目的学习（见图 6-9）
何时使用	当在已有系统之上构建新的设施时使用；当开发涉及多个团队，每个团队负责一个层次上的功能时使用；当存在多个层次上的信息安全需求时使用
优点	只要接口保持不变，允许对整个层进行替换。可以在每个层次上提供冗余设施（例如，身份认证）以增加系统的可依赖性
缺点	在实践中，将各层之间干净地分离经常很难做到，较高层次上的层可能不得不与较低层次上的层直接交互，而不是通过紧邻着的下一层。性能可能是一个问题，因为当一个服务请求在各个层次上进行处理时需要经过多层的解析

图 6-7 分层体系结构模式

图 6-8 是一个包括 4 层的分层体系结构的例子。最底下那层包括系统支持软件，通常是数据库和操作系统支持。上一层是应用层，其中包括关注应用功能的构件以及由其他应用构件使用的公用构件。

从下向上数第三层关注用户界面管理以及提供用户身份认证和授权，而最顶层提供用户界面设施。当然，层的数量是任意的。图 6-8 中的任意一层都可以进一步划分为两层或更多层。

图 6-9 显示第 1 章中所介绍的 iLearn 数字化学习系统有一个遵循这一模式的 4 层体系结构。图 6-19（6.4 节，其中显示了 Mentcare 系统的组织）中显示了另一个分层体系结构模式的例子。

6.3.2 知识库体系结构

作为体系结构模式的例子，分层体系结构和 MVC 模式所呈现的视图都是一个系统的概念化组织。而下一个体系结构模式的例子，知识库模式（图 6-10），则描述了一组相互交互的构件如何共享数据。

大多数使用大量数据的系统都是围绕一个共享数据库或知识库组织的。因此，这个模型适合于数据由一个构件生成同时由另一个构件使用的应用。这类系统的例子包括指挥控制系统、管理信息系统、计算机辅助设计系统、交互式软件开发环境。



图 6-8 一个通用的分层体系结构



图 6-9 iLearn 系统的体系结构

名 称	知识库
描述	一个系统中的所有数据在所有系统构件都能访问的中心知识库中进行管理。构件相互之间并不直接交互，仅通过知识库进行交互
例子	图 6-11 是一个集成开发环境（IDE）的例子，其中构件使用系统设计信息的知识库。每个软件工具生成的信息都会提供给其他工具使用
何时使用	当系统生成大量需要长时间保存的信息时应当使用这个模式。还可以在数据驱动的系统中使用，这类系统中的知识库中增加新数据时会触发一个动作或工具
优点	构件可以保持独立，它们不需要知道其他构件的存在；一个构件进行的修改可以被传播到所有构件；所有数据都可以一致地进行管理（例如同时进行备份），因为这些数据都位于一个地方
缺点	知识库可能存在单点失效问题，因此知识库中的问题会影响整个系统；通过知识库组织所有的通信可能效率不高；将知识库分布到多台计算机上可能比较困难

图 6-10 知识库模式

图 6-11 描述了一种可能的知识库使用情形。图中显示了一个集成开发环境，其中包含不同的支持模型驱动开发的工具。这里的知识库可以是一个追踪对软件的修改并且允许回滚到早期版本的版本控制环境（见第 25 章）。

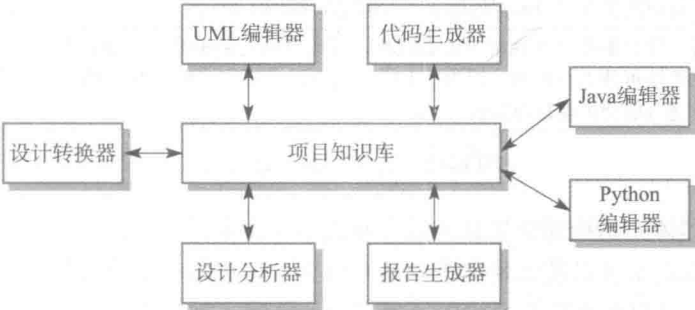


图 6-11 面向集成开发环境（IDE）的知识库体系结构

围绕一个知识库组织工具是一个共享大量数据的有效方法。不需要显式地从一个构件向另一个构件传输数据。然而，构件必须围绕一个达成共识的知识库数据模型运转。这其中不可避免地要对每个工具的特定需要进行权衡，而且如果一个新构件与数据模型不相符，那么就可能很难甚至无法集成了。在实践中，可能很难将知识库分布到多台不同的机器上。虽然有可能实现一个逻辑上的集中知识库的分布式部署，这其中会涉及维护数据的多份拷贝。保证这些拷贝的一致性以及及时更新给系统增加了更多的额外负担。

在图 6-11 所示的知识库体系结构中，知识库是被动的，使用知识库的构件负责进行控制。另一种从人工智能（AI）系统中派生出来的方法是，使用一个“黑板”模型在特定的数据可用时触发构件运行。当知识库中的数据是非结构化数据时，这种方法是合适的。关于应当激活哪个工具的决策只能在对数据进行分析之后做出。这个模型是由 Nii（Nii 1986）引入的，Bosch（Bosch 2000）针对这种风格与系统质量属性的关系进行了很好的讨论。

6.3.3 客户－服务器体系结构

知识库模式关注系统的静态结构，没有显示系统的运行时组织。本节体系结构模式的例子——客户－服务器模式（见图 6-12），描述了一种常用的分布式系统运行时组织方式。一

个遵循客户 - 服务器模式的系统被组织为一组服务和相关联的服务器, 以及访问并使用服务的客户端。该模型的主要构件包括以下这些。

1. 一组向其他构件提供服务的服务器。服务器的例子包括: 提供打印服务的打印服务器, 提供文件管理服务的文件服务器, 提供编程语言编译服务的编译服务器。服务器是软件构件, 多个服务器可能运行在同一台计算机上。
2. 一组调用服务器提供的服务的客户端。通常会有一个客户端程序的多个实例并行运行在多台计算机上。
3. 一个允许客户端访问这些服务的网络。客户 - 服务器系统通常实现为分布式系统, 使用互联网协议连接。

名 称	客户 - 服务器
描述	在客户 - 服务器体系结构中, 系统被呈现为一组服务, 其中每个服务都由一个独立的服务器提供。客户端是这些服务的用户, 通过访问服务器来利用这些服务
例子	图 6-13 是一个组织成客户 - 服务器系统的电影和视频 /DVD 库的例子
何时使用	当一个共享数据库中的数据必须从一系列不同的位置进行访问时使用。因为服务器可以复制, 因此也可以在一个系统的负载多变的情况下使用
优点	这种模型的主要优点是服务器可以分布在网络上。通用的功能 (例如一个打印服务) 可以向所有客户端开放使用, 不需要由所有服务实现
缺点	每个服务都存在单点失效问题, 因此容易受到拒绝服务攻击或服务器失效的影响; 性能可能不可预测, 因为这取决于网络以及系统; 如果服务器属于不同的组织, 那么还可能会出现管理问题

图 6-12 客户 - 服务器模式

客户 - 服务器体系结构通常被认为是分布式系统体系结构, 但是运行在不同的服务器上的独立服务的逻辑模型可以被实现在单台计算机上。这里的一个重要优势仍然是分离和独立性。服务和服务器可以在不影响系统的其他部分的情况下被修改。

客户端可能必须知道可用的服务器以及它们所提供的服务的名称。然而, 服务器不需要知道客户端的身份或者有多少客户端正在访问它们的服务。客户端通过使用请求 - 应答协议 (例如 http) 的远程过程调用访问一个服务器提供的服务, 其中客户端向一个服务器发出请求, 并且一直等待直到收到来自该服务器的应答。

图 6-13 是一个基于客户 - 服务器模型的系统的例子, 这是一个提供电影和照片库的多用户、基于 Web 的系统。在这个系统中, 多个服务器管理并显示不同类型的媒体。视频帧需要以同步的方式快速传输, 但分辨率可以相对较低。它们可以压缩存储在库中, 因此视频服务器可以以不同的格式处理视频压缩和解压缩。然而图片则必须以一种高分辨率保存, 因此将它们保存在一个单独的服务器上更为适合。

目录必须能够处理多种不同的查询, 并提供到 Web 信息系统的链接, 包括关于电影和视频片段的数据, 以及一个支持照片、电影、视频片段销售的电子商务系统。客户端程序只是一个用于访问这些服务的集成用户界面, 使用一个 Web 浏览器来构造。

客户 - 服务器模型最重要的优势在于它是一个分布式体系结构。带有很多分布式处理器的网络化系统可以获得有效的使用。很容易增加一个新的服务器并将其与系统的剩余部分相集成, 或者在不影响系统其他部分的情况下透明地升级服务器。第 17 章介绍分布式体系结构, 其中更详细地解释了客户 - 服务器模型及其变体。

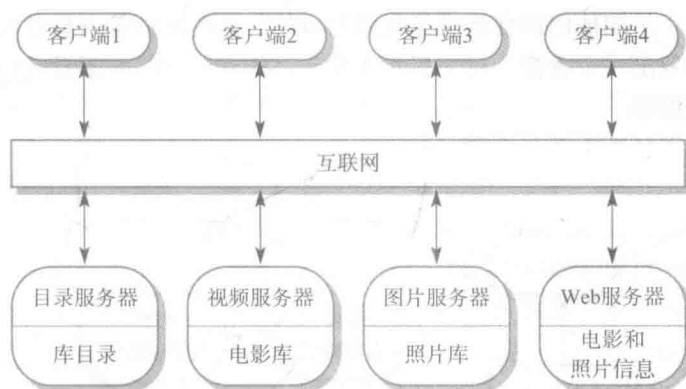


图 6-13 一个电影库的客户-服务器体系结构

6.3.4 管道和过滤器体系结构

最后一个通用体系结构模式的例子是管道和过滤器模式（图 6-14）。这是一个系统的运行时组织结构的模型，其中功能性的变换处理输入并产生输出。数据从一个构件流动到另一个构件，在流经这个序列时进行变换。每个处理步骤被实现为一个变换。输入数据流经这些变换直到被转换为输出。这些变换可以串行或并行执行。每个变换可以一项项地处理数据，也可以在单个批次中一次性处理。

名 称	管道和过滤器
描述	系统中的数据处理通过组织，每个处理构件（过滤器）可以分离开来并执行一种数据转换。数据从一个构件流动（就像在管道中一样）到另一个构件进行处理
例子	图 6-15 是一个用于处理发货单的管道和过滤器系统的例子
何时使用	在数据处理应用（无论是批处理还是基于事务）中得到了广泛应用，这类应用中输入在多个分离的阶段中进行处理，并最终生成相关的输出
优点	容易理解并且支持变换的复用；这种工作流风格与许多业务过程的结构相匹配；通过增加变换来进行演化是非常直观的；可以被实现为一个顺序系统或一个并发系统
缺点	针对数据转换的格式必须在相互通信的多个变换之间达成一致；每个变换必须解析它的输入，并将输出转换成所约定的形式，增加了系统的负担，同时可能意味着无法复用使用不兼容的数据结构的体系结构构件

图 6-14 管道和过滤器模式

“管道和过滤器”这个名字来自于最初的 Unix 系统。在 Unix 中可以使用“管道”将进程连接起来。这些管道将文本流从一个进程传递给另一个进程。符合这一模型的系统可以通过结合 Unix 命令、使用 Unix shell 的管道和控制设施来实现。使用术语过滤器是因为一个转换“过滤掉”来自它的输入数据流中的可以处理的数据。

自从计算机最初被用于自动化地数据处理时，就已经使用该模式的变体了。当变换是顺序化的而所处理的是批量数据时，这个管道和过滤器体系结构模型就成为批量顺序模型，一种通用的数据处理系统（如账单系统）的体系结构。一个嵌入式系统的体系结构也可以被组织为一个进程管道，其中每一个进程并发执行。第 21 章将介绍这种模式在嵌入式系统中的使用。

图 6-15 描述了一个用于批处理应用的这种类型的系统体系结构的例子。一个组织向客

户发出发货单。每个星期他们都会将已经收到的付款与这些发货单进行对账。对于已经付款的发货单，他们会发出一个收据。对于那些在所允许的支付时间内还没有付款的发货单，他们会发出一个付款提醒。

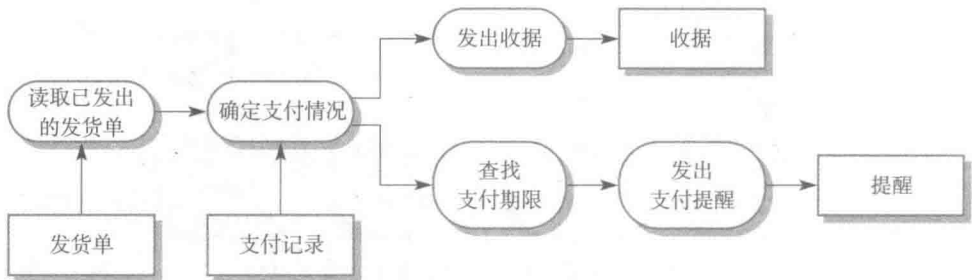


图 6-15 一个管道和过滤器体系结构的例子

管道和过滤器系统最适合于用户交互很少的批处理系统和嵌入式系统。交互式系统很难使用管道和过滤器模型实现，因为管道和过滤器模型需要处理数据流。虽然简单的文本输入和输出可以用这种方式建模，但是对于图形化的用户界面、更复杂的输入/输出格式以及基于事件（例如鼠标点击或菜单选取）的控制策略，很难将这些实现为一种符合管道和过滤器模型的顺序流。



面向控制的体系结构模式

有些特定的体系结构模式反映了通常使用的系统控制结构的组织方式，其中包括集中式的控制（一个构件调用其他构件）以及基于事件的控制（系统对外部事件做出响应）。

<http://software-engineering-book.com/web/archpatterns/>

6.4 应用体系结构

应用系统的目的是满足一个企业或者一个组织的需要。所有的企业都有很多共性：它们都需要雇人、开发货单、保存账户等。同属一个行业的企业使用通用的行业特定的应用。因此，与通用的企业功能一样，所有的电话公司都需要系统来连接呼叫和计费、管理他们的网络、向客户发出账单。因此，这些企业使用的应用系统同样有很多共性。

这些共性促使人们去开发描述特定类型软件系统的结构和组织的软件体系结构。应用体系结构封装了一类系统的主要特性。例如，一个实时系统中可能会包含不同系统类型的通用体系结构模型，如数据收集系统或监控系统。虽然这些系统的具体实例在细节方面存在差异，但是它们的共性体系结构可以在开发同一类型的新系统时被复用。

应用体系结构可以在开发新系统时重新实现。然而，对于许多企业系统，当通过配置通用的应用系统来创建一个新应用时，应用体系结构复用是隐式的。这一点在广泛使用的企业资源规划（Enterprise Resource Planning, ERP）系统以及可配置的成品应用系统（例如，财务和仓库管理）中可以看到。这些系统有一个标准的体系结构以及相关的构件。这些构件可以通过配置和适应性调整来创建一个特定的企业应用。例如，一个供应链管理系统可以通过

适应性调整来支持不同类型的供应商、商品、合同关系。

一个软件设计者可以通过以下这些不同的方式使用应用体系结构模型。

1. 作为体系结构设计过程的起点。如果你不熟悉你正在开发的应用类型，你可以将你的初始设计建立在一个通用的应用体系结构基础上。在此基础上你就可以针对所开发的特定系统对体系结构进行特化。

2. 作为一个设计检查表。如果你已经为一个应用系统开发了一个体系结构设计，那么你可以将它与通用的应用体系结构相比较，检查你的设计是否与通用体系结构相一致。

3. 作为组织开发团队工作的一种方式。应用体系结构识别出了系统体系结构中稳定的结构特征，而且这些结构单元很多时候可以并行开发。你可以按照这一结构向团队成员分配工作，让他们实现体系结构中的不同构件。

4. 作为评价构件复用的一种方式。如果你有一些可复用的构件，可以将它们与通用结构相比较，以确定应用体系结构中是否有与之相当的构件。

5. 作为谈论应用时的一种词汇表。如果你正在讨论一个特定的应用或者试图比较不同的应用，那么你可以使用通用体系结构中所识别的概念来讨论这些应用。

存在许多类型的应用系统，有时候它们可能会看上去很不一样。然而，表面上不一样的应用可能有很多共性，因此可以共享一个抽象应用体系结构。这里将通过描述两种不同类型的应用的体系结构来说明这一点。

1. 事务处理应用。事务处理应用是以数据库为中心的应用，处理用户的信息请求并且更新数据库中的信息。这些系统是最常见的交互式业务系统类型。这些系统的组织结构使用户动作不会相互干涉，而数据库的完整性得以保持。这类系统包括交互式银行系统、电子商务系统、信息系统、预订系统。

2. 语言处理系统。语言处理系统中用户的意图用形式化语言（例如编程语言）来表达。语言处理系统将这种语言处理成一种内部格式，然后对这种内部表示进行解释。最著名的语言处理系统是编译器，它将高层语言程序转换为机器代码。然而，语言处理系统也可以用于解释数据库和信息系统的命令语言以及标记语言（例如 XML）。

这里选择这两类特定类型的系统是因为，大量基于 Web 的业务系统都是事务处理系统，而所有的软件开发都依赖于语言处理系统。



应用体系结构

在本书的网站上有几个应用体系结构的例子，其中包括批量数据处理系统、资源分配系统、基于事件的编辑系统的描述。

<http://software-engineering-book.com/web/apparch/>

6.4.1 事务处理系统

事务处理系统被设计用于处理用户获取数据库中信息的请求或者更新数据库的请求（Lewis, Bernstein, and Kifer 2003）。从技术上看，一个数据库事务包含一个操作序列并且该序列作为整体处理（一个原子单元）。一个事务中的所有操作必须在数据库修改被持久化

之前完成。这保证了一个事务中失败的操作不会导致数据库中的不一致。

从用户的角度看，一个事务是任何满足一个目标的内聚的操作序列，例如“查找从伦敦到巴黎的航班时间”。如果用户事务不需要对数据库进行修改，那么不一定要将其作为一个技术上的数据库事务。

一个数据库事务的例子是，一个客户使用一台 ATM 机请求从一个银行账户中取一些钱。这包括检查客户账户余额以确定是否有足够的钱、按照取款金额修改余额、发送命令给 ATM 机以提供现金。在所有这些步骤完成之前，这个事务都没有完成并且客户账户数据库不会进行修改。

事务处理系统通常是交互式系统，其中用户发出异步的服务请求。图 6-16 描述了事务处理应用的概念体系结构。首先，用户通过一个输入 / 输出处理构件向系统发出请求。该请求由一些应用特定的逻辑进行处理。一个事务被创建并传递给事务管理器（通常嵌入在数据库管理系统中）。事务管理器在保证事务被正确完成后，再通知应用处理已完成。

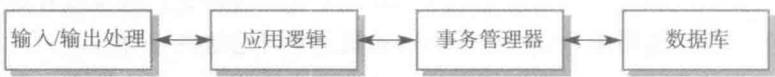


图 6-16 事务处理应用的结构

可将事务处理系统组织为含有负责输入、处理、输出的系统构件的“管道和过滤器”体系结构。例如，考虑一个允许客户用 ATM 机查询账户以及取现金的银行系统。该系统由两个协作的软件构件组成——ATM 软件以及银行的数据库服务器中的账户处理软件。输入和输出构件被实现为 ATM 中的软件，而处理构件则是银行的数据库服务器的一部分。图 6-17 描述了这个系统的体系结构，显示了输入、处理、输出构件的功能。

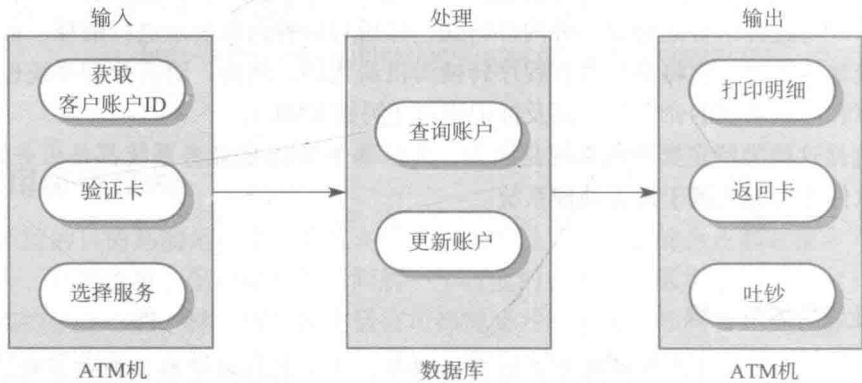


图 6-17 一个 ATM 系统的软件体系结构

6.4.2 信息系统

所有包含与一个共享数据库交互的系统都可以认为是基于事务的信息系统。一个信息系统允许对一个很大的信息库（例如，图书馆目录、航班时间表或者医院里的病人记录）的受控访问。信息系统几乎都是基于 Web 的系统，其中用户界面在一个 Web 浏览器中实现。

图 6-18 展示了一个非常通用的信息系统模型。系统使用层次化的方法（见 6.3 节）进行建模，其中最顶层支持用户界面，最底层是系统数据库。用户通信层处理来自用户界面的所

有输入和输出，信息检索层包括特定应用的数据库访问和更新逻辑。这个模型中的各个层可以直接映射到一个分布式基于互联网的系统中的服务器上。

作为这一分层模型的一个实例化的例子，图 6-19 描述了 Mentcare 系统的体系结构。该系统维护并管理正在接受专科医生关于心理健康问题诊疗的病人的详细信息。通过识别支持用户通信以及信息检索和访问的构件，向这个模型的每一层中都增加了一些细节。

1. 最顶层是一个基于浏览器的用户界面。

2. 第二层实现了通过 Web 浏览器提供的用户界面功能：允许用户登录系统的构件，确保用户使用的操作符合其角色权限的检查构件。这一层包括：向用户呈现信息的表单以及菜单管理构件，检查信息一致性的数据验证构件。

3. 第三层实现了系统的功能，所提供的构件实现了系统信息安全、病人信息创建和更新、从其他数据库导入和导出病人数据、创建管理报表的报表生成器。

4. 最终，最底层使用一个商业化的数据库管理系统来构建，提供了事务管理和持久化的数据存贮。

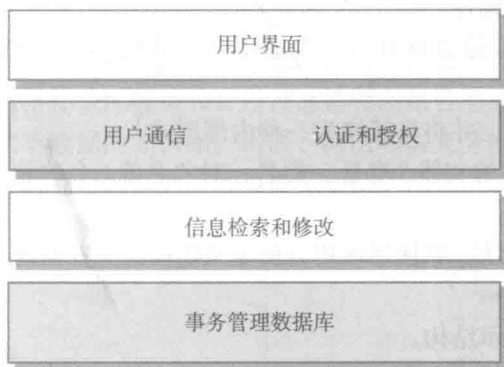


图 6-18 分层的信息系统体系结构

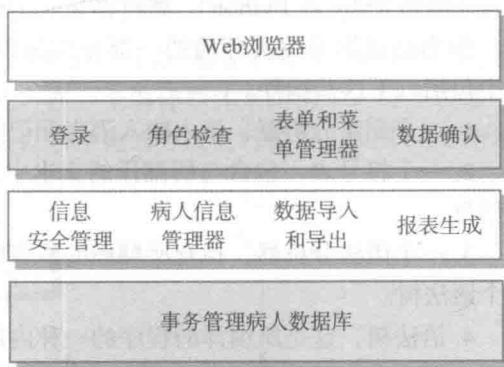


图 6-19 Mentcare 系统的体系结构

信息和资源管理系统有时候也是事务处理系统。例如，电子商务系统是基于互联网的资源管理系统，它们接受商品或服务的电子订单，然后安排这些商品或服务向客户交付。在一个电子商务系统中，应用特定层包括附加的功能以支持“购物车”，其中用户可以通过多个分开的事务放入一些购物项，然后在一个事务中一起支付所有购物项。

这些系统中的服务器组织通常反映了图 6-18 中所示的 4 层通用模型。这些系统经常实现为带有多层客户端 / 服务器体系结构的分布式系统。

1. Web 服务器负责所有的用户通信，并带有使用 Web 浏览器实现的用户界面。

2. 应用服务器负责实现特定应用逻辑以及信息存储和检索请求。

3. 数据库服务器将信息移动到数据库，从数据库中获取数据，同时处理事务管理。

使用多个服务器可以实现高吞吐量，使每分钟处理成千上万的事务成为可能。随着请求量的增长，每个层次都可以增加服务器以应对所需要的额外处理能力。

6.4.3 语言处理系统

语言处理系统将一种语言翻译为该语言的另一种表示方式，对于编程语言还可以执行所产生的代码。编译器将一种编程语言翻译为机器代码。其他语言处理系统可以将 XML 数据描述翻译为数据库查询命令或者另一种 XML 表示形式。自然语言处理系统可以将一种自然

语言翻译为另一种语言，例如从法语到挪威语。

一个面向编程语言的语言处理系统体系结构如图 6-20 所示。源语言指令定义了要执行的程序，一个翻译器将这些转换为面向抽象机器的指令。这些指令接着被另一个构件解析，该构件取出执行指令并且使用（如果有必要的话）来自环境的数据执行这些指令。该过程的输出是解析输入数据中的指令的结果。

对于许多编译器，解析器是处理机器指令的系统硬件，而抽象机器是真实的处理器。然而，对于动态类型语言（例如 Ruby 或 Python），解析器是一个软件构件。

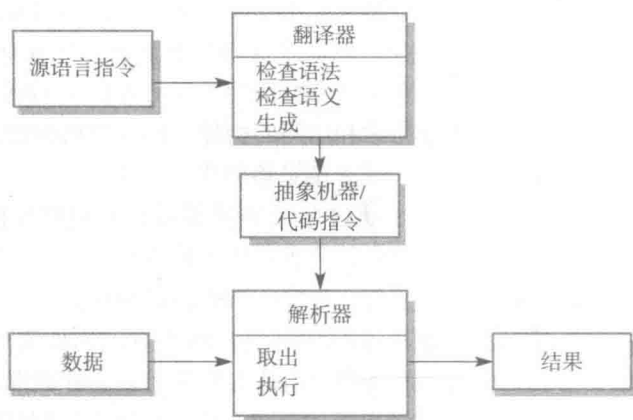


图 6-20 一个语言处理系统的体系结构

作为更通用的编程环境的一部分的编程语言编译器具有一个通用体系结构（图 6-21），其中包括以下这些构件。

1. 一个词法分析器，读入输入语言标记符号，并将其转换为一种内部形式。
2. 一个符号表，包含与所翻译的文本中使用的实体（变量、类名、对象名等）名称相关的信息。
3. 一个语法分析器，检查所翻译的语言的语法。它使用该语言所定义的语法，并且构造一个语法树。
4. 语法树，这是所编译的程序的一种内部表示结构。
5. 语义分析器，使用来自语法树的信息以及符号表来检查输入语言文本的语义正确性。
6. 代码生成器，遍历语法树并且生成抽象的机器代码。

还可以包含其他对语法树进行分析和转换以提高效率并从所生成的机器代码中消除冗余的构件。在其他类型的语言处理系统（例如，一个自然语言翻译器）中，还有一些附加的构件，例如字典。系统的输出是对输入文本的翻译。



参考体系结构

参考体系结构捕捉一个领域中系统体系结构的重要特征。从本质上讲，它们包括一个应用体系结构可能包含的所有东西，然而在现实中任何单个的应用都不太可能包含参考体系结构中所显示的所有特征。参考体系结构的主要目的是评价和比较设计方案，以及对相关人员进行该领域中的体系结构特性的培训。

<http://software-engineering-book.com/web/refarch/>

图 6-21 描述了一个语言处理系统如何作为一个集成的编程支持工具集的一部分。在这个例子中，符号表和语法树扮演中心信息库的角色。工具或者工具片段通过这个库进行通信。有时可以嵌入在工具中的其他信息（例如程序的语法定义以及输出格式定义）从工具中

都被抽取出来并放到知识库中。因此,一个语法制导的编辑器可以在一个程序正在编写时检查它的语法是否正确。一个程序格式器可以创建高亮显示不同语法元素的清单,从而使程序更容易阅读和理解。

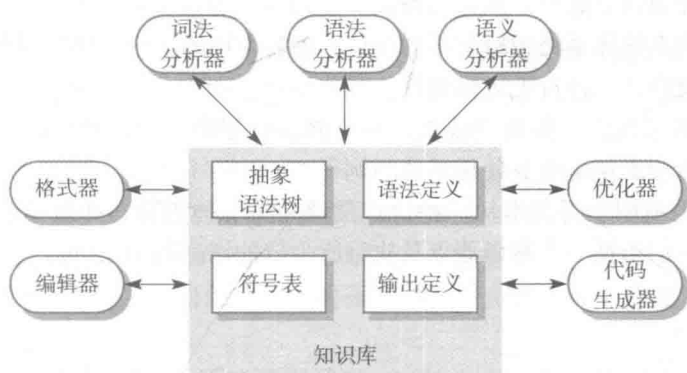


图 6-21 一个语言处理系统的知识库体系结构

其他体系结构模式也可以用于语言处理系统 (Garlan and Shaw 1993)。编译器的实现可以将知识库模型和管道过滤器模型结合起来使用。在一个编译器体系结构中,符号表是一个共享数据的知识库。词法、语法和语义分析阶段以顺序化的方式进行组织 (如图 6-22 所示),并且通过共享的符号表进行通信。

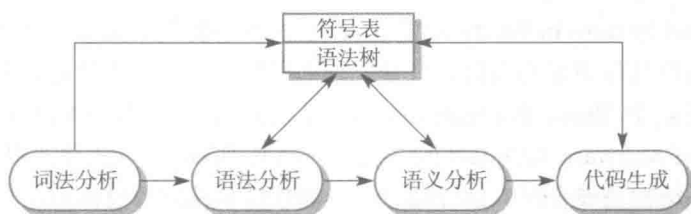


图 6-22 一个管道和过滤器编译器体系结构

这一语言编译的管道和过滤器模型在批处理环境中很有效,其中程序的编译和执行都不需要用户交互,例如,将一个 XML 文档翻译成另一个 XML 文档时。但是,当一个编译器与其他语言处理工具 (例如,一个结构化编辑系统、一个交互式调试器或者一个程序格式器) 相集成时,该模型就没那么有效了。在这种情形中,来自一个构件的变化需要立即反映在其他构件中。如果你正在实现一个通用的、面向语言的编程环境,那么一个更好的办法是围绕一个知识库来组织系统,如图 6-21 所示。

要点

- 一个软件体系结构是关于软件系统如何组织的一种描述。一个系统的属性,例如性能、信息安全、可用性等,都受到所使用的体系结构的影响。
- 体系结构设计决策包括对应用类型、系统分布、要使用的体系结构风格、体系结构的描述和评价方式等方面的决策。
- 体系结构可以从多个不同的视角或视图来描述。可能的视图包括概念视图、逻辑视图、进程视图、开发视图、物理视图。

- 体系结构模式是一种复用关于通用系统体系结构的知识的手段。它们描述了体系结构，解释应该何时使用，还指出了它的优点和缺点。
- 广泛使用的体系结构模式包括模型-视图-控制器（MVC）模式、分层体系结构模式、知识库模式、客户-服务器模式、管道和过滤器模式。
- 通用的应用系统体系结构模型帮助我们：理解应用的运行，比较同种类型的应用，验证应用系统设计，评价大规模构件的可复用性。
- 事务处理系统允许一些用户远程访问并修改数据库中的信息的交互式系统。信息系统和资源管理系统是事务处理系统的例子。
- 语言处理系统用于将文本从一种语言翻译为另一种语言，并执行输入语言中所指定的指令。它们包括一个翻译器以及执行所生成的语言的抽象机器。

阅读推荐

《Software Architecture: Perspectives on an Emerging Discipline》是第一本关于软件体系结构的书，对于目前仍然关注的一些不同的体系结构风格进行了很好的讨论。（M. Shaw and D. Garlan, 1996, Prentice-Hall）

《The Golden Age of Software Architecture》这篇论文对软件体系结构的发展进行了综述，从20世纪80年代开始提出这一概念直到21世纪软件体系结构的使用。其中没有太多的技术内容，但是提供了一个很有趣的历史回顾。（M. Shaw and P. Clements, IEEE Software, 21（2），March-April 2006）<http://doi.dx.org/10.1109/MS.2006.58>

《Software Architecture in Practice（3rd ed.）》这本书提供了关于软件体系结构的实践讨论，其中没有过分吹嘘体系结构设计的好处。该书提供了一个清晰的企业原理，解释了体系结构为什么很重要。（L. Bass, P. Clements, and R. Kazman, 2012, Addison-Wesley）

《Handbook of Software Architecture》是由Grady Booch（软件体系结构的早期传道者之一）所写的体系结构进展介绍。他曾经对一系列软件系统进行过体系结构文档化，因此可以看到一些现实而非学术界的抽象。这份文献在Web上，不久的将来将出版。（G. Booch, 2014）<http://www.handbookofsoftwarearchitecture.com/>

网站

本章的PPT：<http://software-engineering-book.com/slides/chap6/>

支持视频的链接：<http://software-engineering-book.com/videos/requirements-and-design/>

练习

- 6.1 当描述一个系统时，为什么必须要在得到完整的需求规格说明之前就开始系统体系结构的设计？
- 6.2 你被要求准备并向一个非技术管理人员做陈述，说明为一个新项目雇用系统架构师的理由。将其中的要点一一列举出来，说明你的陈述中的关键点，其中要解释软件体系结构的重要性。
- 6.3 在为系统的一个可用性和信息安全需求都是最重要的非功能性需求的系统设计体系结构时，为什么可能会出现设计冲突？
- 6.4 针对以下这些系统，画出描述它们的体系结构的一个概念视图和一个进程视图。

- 一个火车站里由乘客使用的售票机；
- 一个计算机控制的视频会议系统，该系统允许多个与会者同时看到相关的视频、音频和计算机数据；
- 一个用于清扫相对清楚的区域（例如走廊）的扫地机器人，必须能够感知墙和其他障碍物。
- 6.5 为什么你在设计一个大型系统的体系结构时通常要使用多个体系结构模式？
- 6.6 针对一个用于在互联网上销售和分发音乐的系统（例如 iTunes），建议一个体系结构。你所提出的体系结构是以哪些体系结构模式为基础的？
- 6.7 将要开发一个信息系统以用于维护关于一个公用事业公司所拥有资产（例如建筑、车辆、设备）的信息。希望该系统可以在新的资产信息可用时，在现场工作的员工可以使用移动设备进行更新。该公司有几个已有的资产数据库，它们应当通过该系统进行集成。基于图 6-18 中所示的通用信息系统体系结构，为这个资产管理系统设计一个分层体系结构。
- 6.8 使用这里所介绍的语言处理系统通用模型，设计一个系统的体系结构，该系统接受自然语言命令，并将其翻译为数据库查询语言（例如 SQL）。
- 6.9 使用如图 6-18 中所示的信息系统基本模型，针对一个面向移动设备用于显示某个特定机场航班到达和起飞信息的应用，建议其中应该包含哪些构件？
- 6.10 是否应当设置一个独立的“软件架构师”职业，其角色是与客户一起独立工作来设计软件系统的体系结构？一个独立的软件公司接下来可以实现这个系统。设置这样一种职业可能会存在什么困难？

参考文献

- Bass, L., P. Clements, and R. Kazman. 2012. *Software Architecture in Practice (3rd ed.)*. Boston: Addison-Wesley.
- Berczuk, S. P., and B. Appleton. 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley.
- Booch, G. 2014. "Handbook of Software Architecture." <http://handbookofsoftwarearchitecture.com/>
- Bosch, J. 2000. *Design and Use of Software Architectures*. Harlow, UK: Addison-Wesley.
- Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.
- . 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.
- Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.
- Chen, L., M. Ali Babar, and B. Nuseibeh. 2013. "Characterizing Architecturally Significant Requirements." *IEEE Software* 30 (2): 38–45. doi:10.1109/MS.2012.174.
- Coplien, J. O., and N. B. Harrison. 2004. *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice-Hall.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

- Garlan, D., and M. Shaw. 1993. "An Introduction to Software Architecture." In *Advances in Software Engineering and Knowledge Engineering*, edited by V. Ambriola and G. Tortora, 2:1-39. London: World Scientific Publishing Co.
- Hofmeister, C., R. Nord, and D. Soni. 2000. *Applied Software Architecture*. Boston: Addison-Wesley.
- Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.
- Krutchén, P. 1995. "The 4+1 View Model of Software Architecture." *IEEE Software* 12 (6): 42-50. doi:10.1109/52.469759.
- Lange, C. F. J., M. R. V. Chaudron, and J. Muskens. 2006. "UML Software Architecture and Design Description." *IEEE Software* 23 (2): 40-46. doi:10.1109/MS.2006.50.
- Lewis, P. M., A. J. Bernstein, and M. Kifer. 2003. *Databases and Transaction Processing: An Application-Oriented Approach*. Boston: Addison-Wesley.
- Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59-89. doi:10.1145/972648.972651.
- Nii, H. P. 1986. "Blackboard Systems, Parts 1 and 2." *AI Magazine* 7 (2 and 3): 38-53 and 62-69. <http://www.aaai.org/ojs/index.php/aimagazine/article/view/537/473>
- Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.
- Shaw, M., and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall.
- Usability Group. 1998. "Usability Patterns". University of Brighton. <http://www.it.bton.ac.uk/Research/patterns/home.html>

设计和实现

目标

本章的目标是介绍使用 UML 的面向对象软件设计，并强调一些重要的实现关注点。读完本章后，你将：

- 理解一个通用的面向对象设计过程中最重要的活动；
- 理解一些不同的面向对象设计描述模型；
- 了解设计模式的思想，以及如何利用设计模式实现设计知识和经验的复用；
- 了解在实现软件时必须考虑的一些关键问题，包括软件复用和开源开发。

软件设计和实现是软件工程过程中的一个阶段，在此阶段中会开发一个可执行的软件系统。对于一些简单的系统，软件工程意味着软件设计和实现，而所有其他软件工程活动都被合并到这个过程中了。然而，对于大型系统而言，软件设计和实现只是一系列软件工程过程（例如，需求工程、验证和确认等）中的一个。

软件设计和实现活动一般总是存在重叠。软件设计是一个创造性的活动，在此活动中需要基于客户需求识别软件构件及其关系。实现是将设计实现为一个程序的过程。有时候会有一个独立的设计阶段，其中的设计会被建模和文档化。而其他时候设计会存在于程序员的脑海中，或只是在白板或纸上大致地用草图画一下。设计是关于如何解决一个问题，因此总是有一个设计过程。然而，并不总是有必要或者适合用 UML 或其他设计描述语言详细描述设计。

设计和实现密切联系在一起，考虑一个设计时通常都应该将实现问题考虑进来。例如，如果使用面向对象语言（例如 Java 或 C#）编程，那么使用 UML 描述一个设计可能是正确的选择。如果使用 Python 这样的动态类型语言，那么 UML 就没那么有用了。如果通过配置一个成品软件包来实现系统，那么使用 UML 就没什么道理了。如第 3 章中所提到的，敏捷方法通常是在非正式的设计草图基础上工作的，设计决策都留给了程序员。

在一个软件项目的早期阶段必须做出的一个最重要的实现决策是，确定自己构建还是购买应用软件。对于许多类型的应用而言，现在都可以买到可以按照用户需求进行适应性调整和裁剪的成品应用系统。例如，如果想实现一个医疗记录系统，可以购买一个已经在医院中使用的软件包。采用这种方法通常比用传统开发语言开发一个新系统更加便宜也更快。

当通过复用一个成品软件产品开发一个应用系统时，设计过程关注如何配置该系统产品以满足应用需求。你不需要开发系统的设计模型，例如，关于系统对象及其交互的模型。第 15 章中将介绍这一基于复用的开发方法。

这里假设本书的大部分读者都已经有了程序设计和实现经验。这些都是你在学习编程以及掌握一个编程语言（例如 Java 或 Python）的元素时可以获取到的。通过学习编程语言，你一般都可以学到你所学习的编程语言中的一些好的编程实践，以及学会如何调试你所

开发的程序。因此，在这里不再介绍编程相关的话题。本章有以下两个目的：

1. 展示系统建模和体系结构设计（在第 5 章和第 6 章中介绍）如何在开发一个面向对象软件设计时进行实践应用；
2. 介绍在编程相关的书中通常不会涉及的一些重要的实现问题，包括软件复用、配置管理和开源开发。

由于存在大量不同的开发平台，本章不会特别关注任何特定的编程语言或实现技术。因此，本章中所有的例子都是用 UML 而不是编程语言（例如 Java 或 Python）来呈现的。

7.1 使用 UML 的面向对象设计

一个面向对象系统由相互交互的对象组成，这些对象保持自己的本地状态同时基于状态提供相应的操作。状态的表示是私有的，无法从该对象的外部直接访问。面向对象设计过程包括设计对象类以及这些类之间的关系。这些类定义了系统中的对象以及它们的交互。当设计被实现为执行程序时，对象会在这些类定义的基础上被动态创建出来。

对象同时包括数据以及操纵这些数据的操作。因此，它们可以作为独立的实体进行理解和修改。改变一个对象的实现或者增加服务不应该影响其他系统对象。因为对象与事物相关联，现实世界实体（例如硬件构件）和它们在系统中的控制对象之间经常存在清晰的映射。这改进了设计的可理解性，因此也有利于可维护性。

为了开发一个系统设计（从概念到详细的面向对象设计），你需要：

1. 理解并定义上下文以及与系统的外部交互；
2. 设计系统体系结构；
3. 识别系统中的主要对象；
4. 开发设计模型；
5. 刻画接口。

像所有的创造性活动一样，设计不是一个清晰的顺序性过程。在开发一个设计的过程中会获得想法，提出解决方案，随着更多信息的获得精化这些解决方案。当出现问题时，不可避免地必须回溯并重新尝试。有时候要详细探索各种选项看它们是否奏效；而其他时候则会忽略细节直到过程中的后期。有时候使用 UML 等建模表示法来精确地阐明设计的各个方面；而其他时候，则可以以非正式的方式使用一些表示法来激发讨论。

本章通过为业务气象站这个嵌入式系统（在第 1 章中介绍过）的一部分开发一个设计来解释面向对象软件设计。每个气象站都会记录本地的气象信息，并使用卫星链路定期将这些信息发送给一个气象信息系统。

7.1.1 系统上下文和交互

任何软件设计过程的第一个阶段都是理解所设计的软件与它的外部环境之间的关系。这是很重要的，因为可以据此决定如何提供所需要的系统功能，以及如何对系统进行组织从而与环境进行通信。如第 5 章中所提到的，理解上下文还可以建立系统的边界。

设定系统边界帮助开发人员决定哪些特征在所设计的系统中实现，以及哪些特征在其他相关联的系统中实现。在野外气象站案例中，需要决定功能如何在所有气象站的控制软件以及气象站自身的嵌入式软件之间进行分布。

系统上下文模型和交互模型所呈现的关于系统及其环境之间关系的视图是互补的。

1. 系统上下文模型是一种结构化模型，其中展示了所开发的系统的环境中的其他系统。

2. 交互模型是一种动态模型，其中显示系统在使用时如何与环境进行交互。

一个系统的上下文模型可以使用关联来表示。关联只是显示关联中所包含的实体之间存在某些关系。可以使用简单的框图来描述系统的环境，显示系统中的实体以及它们之间的关联关系。图 7-1 显示了每个气象站环境中的系统包括一个气象信息系统、一个卫星系统以及一个控制系统。链接关系上的基数信息显示，有一个控制系统、多个气象站、一个卫星、一个通用的气象信息系统。

当对系统与其环境之间的交互进行建模时，应当使用一种不包含太多细节的抽象的方法。其中一种方法是使用用况模型。如第 4 章和第 5 章所述，每个用况表示与系统的一种交互。每种可能的交互都被表示为一个椭圆并进行命名，参与交互的外部实体由一个线条人形来表示。

气象站的用况模型如图 7-2 所示。图中显示气象站与气象信息系统进行交互以报告气象数据以及气象站硬件的状态。此外，还包括与一个控制系统的交互，通过该交互可以发出特定的气象站控制命令。线条人形在 UML 中用于表示其他系统以及人用户。

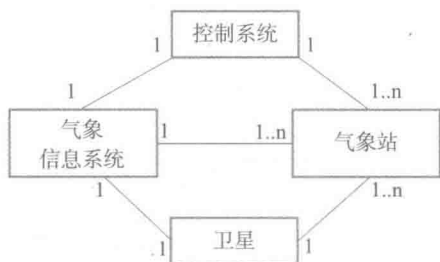


图 7-1 气象站的系统上下文

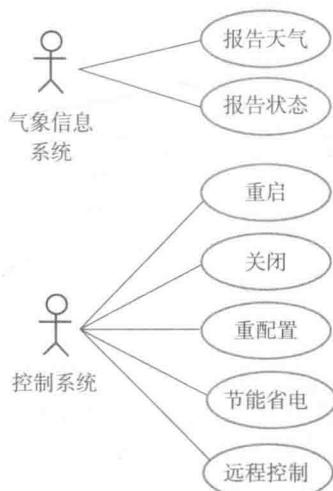


图 7-2 气象站用况



气象站用况

报告天气——发送天气数据到气象信息系统

报告状态——发送状态信息到气象信息系统

重启——如果气象站被关闭了，那么重启系统

关闭——关闭气象站

重配置——重配置气象站软件

省电——将气象站设置为节电模式

远程控制——发送控制命令到任何气象站子系统

<http://software-engineering-book.com/web/ws-use-cases/>

这些用况中的每一个都应当使用结构化自然语言进行描述。这有助于设计人员识别系统中的对象并让他们理解系统意图要做什么。本章会使用一种标准的格式来进行这一描述，其中清晰地识别了要交换什么信息、交互如何发起等。如第 21 章所述，嵌入式系统经常通过

描述它们如何响应内部或外部激励的方式进行建模。因此，激励以及相关的响应应当在描述中进行列举。图 7-3 展示了基于这种方法对图 7-2 中的“报告天气”用况的描述。

系统	气象站
用况	报告天气
参与者	气象信息系统、气象站
数据	气象站将在收集阶段从各种仪器上收集的气象数据的一个汇总发送给气象信息系统。所发送的数据包括最高、最低以及平均的地面温度和空气温度；最高、最低以及平均的气压；最高、最低以及平均的风速；总降雨量；以 5 分钟间隔采样的风向。
激励	气象信息系统与气象站之间建立一个微信通信链路并请求传输数据。
响应	汇总后的数据被发送到气象信息系统。
备注	通常会让气象站每小时报告一次，但不同气象站的报告频率可能会有所差别并且可能在未来发生变化。

图 7-3 用况描述：报告天气

7.1.2 体系结构设计

软件系统和系统环境之间的交互定义好之后，就可以以此为基础来设计系统体系结构。当然，设计人员需要将这一知识与自身关于体系结构设计原则的通用知识以及更加详细的领域知识相结合。需要识别构成系统的主要构件以及它们之间的交互，然后要使用一种体系结构模式（例如，分层或客户 - 服务器模型）来设计系统的组织结构。

气象站软件的高层体系结构设计如图 7-4 所示。气象站由一些独立的子系统组成，包括故障管理器（Fault manager）子系统、配置管理器（Configuration manager）子系统、电源管理器（Power manager）子系统、通信（Communications）子系统、数据收集（Data collection）子系统、仪器（Instruments）子系统。这些子系统通过在公共基础设施上广播消息来进行通信，如图 7-4 中的通信（Communication）链路所示。每个子系统都会在该基础设施上监听消息并选取与它们相关的消息。这一“监听器模型”是一种广泛使用的分布式系统体系结构风格。

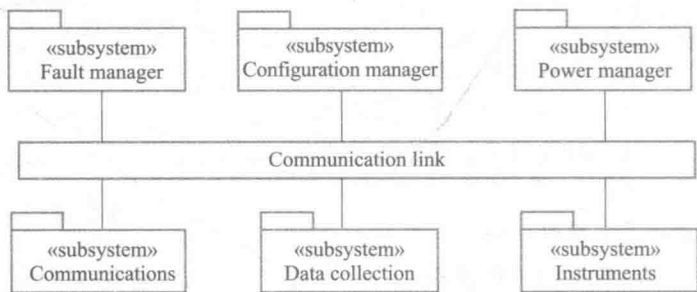


图 7-4 气象站的高层体系结构

当通信子系统收到一个控制命令（例如关闭）时，其他每个子系统也将获得该命令，这些子系统接收者会执行命令（例如，按照正确的方式关闭自身）。这一体系结构的关键优点是很容易支持不同的子系统配置，因为一条消息的发送者不需要指定接收该消息的特定子系统。

图 7-5 展示了图 7-4 中所包含的数据收集子系统的体系结构。其中的发送器（Transmitter）和接收器（Receiver）对象关注通信管理，而气象数据（WeatherData）对象则封装从仪器那

里收集并传送给气象信息系统的信息。这一设计遵循了生产者 - 消费者模式，如第 21 章所述。

7.1.3 对象类识别

在设计过程的这个阶段之前，关于你所设计的系统中所包含的重要对象你应该已经有了一些想法。随着你对设计的理解进一步加深，要对这些关于系统对象的想法进行精化。用况描述帮助你识别系统中的对象和操作。从“报告天气”用况的描述中可以很清楚地看到，你将要实现表示收集气象数据的各种仪器的对象以及一个表示气象数据汇总的对象。通常你还会需要一个高层的系统对象或者用来封装用况中所定义的系统交互的对象。脑海中有了一些对象之后，便可以开始识别系统中的通用对象类。

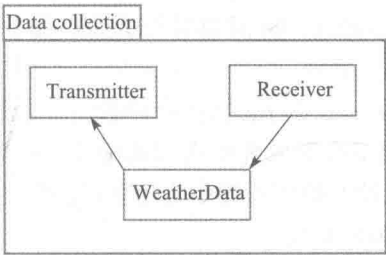


图 7-5 数据收集系统的体系结构

随着 20 世纪 80 年代面向对象设计的发展，出现了不同的识别面向对象系统中对象类的方法，例如下面这些。

- 1. 对所要构建的系统的自然语言描述进行语法分析。对象和属性是名词；操作或服务是动词 (Abbott 1983)。
- 2. 使用应用领域中有形的实体或事物 (例如飞机)、角色 (例如经理)、事件 (例如请求)、交互 (例如会议)、位置 (例如办公室)、组织单元 (例如公司) 等 (Wirfs-Brock, Wilkerson, and Weiner 1990)。
- 3. 使用一种基于场景的分析方法，其中依次识别并分析系统使用的各种场景。分析每个场景时，负责分析的团队必须识别所需要的对象、属性以及操作 (Beck and Cunningham 1989)。

在实践中，你必须使用多种知识源来发现对象类。最初从非正式的系统描述中所识别出的对象类、属性和操作可以成为设计的起点。接下来，来自应用领域知识或场景分析的信息可以用于精化和扩展初始的对象。这些信息可以从需求文档、与用户的讨论或者对现有系统的分析中收集到。除了表示系统外部实体的对象，还必须设计出用于提供通用服务 (例如，搜索和正确性检查) 的“实现对象”。

在野外气象站中，对象识别是基于系统中有形的硬件。由于篇幅原因，这里无法列举所有的系统对象。图 7-6 中展示了 5 个对象类。Ground thermometer (地面温度计)、Anemometer (风速计) 和 Barometer (气压计) 对象是应用领域对象，WeatherStation (气象站) 和 WeatherData (气象数据) 对象是从系统描述和场景 (用况) 描述中识别出来的。

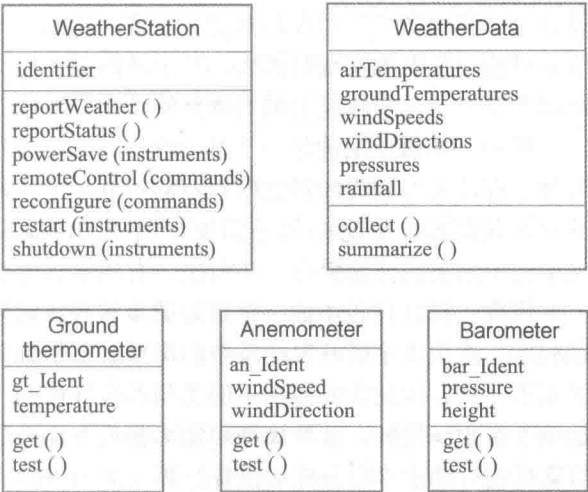


图 7-6 气象站对象

1. WeatherStation 对象类提供了气

象站及其环境的基本接口。它的操作基于图 7-3 中所示的交互。这里使用单个对象类，其中包括所有这些交互。也可以将系统接口设计为多个不同的类，每个类对应一个交互。

2. WeatherData 对象类负责处理报告天气的命令。它将来自气象站仪器的汇总数据发送到气象信息系统。

3. Ground thermometer、Anemometer 和 Barometer 对象类直接与系统中的仪器相关联。它们反映了系统中有形的硬件实体，而它们的操作关注控制这些硬件。这些对象自治地运行，按照指定的频率收集数据，并在本地保存所收集的数据。这些数据在请求时提供给 WeatherData 对象。

可以使用应用领域知识来识别其他对象、属性和服务。

1. 气象站经常位于偏远的地方，而且所部署的各种仪器有时会坏。仪器失效应当自动报告。这意味着你需要用于检查这些仪器是否正确工作的属性和操作。

2. 有许多远程气象站，因此每个气象站应当有自己的标识符以便在通信过程中进行唯一标识。

3. 由于气象站是在不同的时间安装的，仪器的类型可能会不一样。因此，每个仪器也应当被唯一标识，并且应当要维护一个仪器信息数据库。

在这个设计过程阶段中，你应当关注对象自身，不需要考虑这些对象要如何实现。一旦识别出这些对象后，可以再对这些对象设计进行精化。你会发现共性的特征，然后为系统设计继承层次。例如，你会识别出一个 Instrument（仪器）父类，其中定义了所有仪器的共性特征（例如，一个标识符、读取信息操作、测试操作等）。你还可以向父类中增加新的属性和操作，例如，一个记录数据应当多久收集一次的属性。

7.1.4 设计模型

设计或系统模型，如第 5 章中所述，展示了一个系统中的对象或者对象类，还展示了这些实体之间的关联和关系。这些模型是系统需求和系统实现之间的桥梁。它们必须是抽象的，以使得不必要的细节不会隐藏它们和系统需求之间的关系。然而，模型还必须包含足够的细节，以使得程序员可以做出实现决策。

一个设计模型所需要的详细程度取决于开发人员所使用的设计过程。由于需求工程师、设计人员和程序员之间存在密切的联系，抽象模型可能就是所有所需要的东西。特定的设计决策可能会在系统实现时做出，其中的问题通过非正式的讨论来解决。与之相似，如果使用敏捷开发，在一个白板上画出概要的设计模型可能就足以提供所有所需要的信息。

然而，如果使用的是一个基于计划的开发过程，你可能需要更详细的模型。如果需求工程师、设计人员和程序员之间仅有间接联系（例如，一个系统在一个组织的某个部门进行设计但在其他地方实现），那么需要详细的设计描述以便于沟通。此时要使用从高层的抽象模型中派生出来的详细模型，从而使所有的团队成员对于设计有一个共同的理解。

因此，设计过程中的一个重要的步骤是决定所需要的设计模型以及这些模型所需要的详细程度。这取决于所开发的系统的类型。一个顺序性的数据处理系统与一个嵌入式实时系统就很不一样，因此需要使用不同类型的设计模型。UML 支持 13 种不同类型的模型，但是正如第 5 章中所说的，这些模型中很多都没有得到广泛使用。尽量减少所产生的模型的数量可以降低设计的成本以及完成设计过程所需的时间。

当使用 UML 来开发一个设计时，应当开发以下两类设计模型。

1. 结构模型, 使用静态类及其关系描述系统的静态结构。这个阶段需要描述的重要的关系类型包括泛化(继承)关系、使用/被使用关系、组合关系。

2. 动态模型, 描述了系统的动态结构并展示了所期望的系统对象之间的运行时交互。可以描述的交互包括对象发出的服务请求的序列以及由这些对象交互所触发的状态变化。

以下3种UML模型对于增加用况和体系结构模型的细节特别有用。

1. 子系统模型, 展示了如何对对象进行逻辑分组以构成内聚的子系统。这些可以使用一种类图的形式来表示, 其中每个子系统表示为一个包含对象的包。子系统模型是结构模型。

2. 顺序模型, 展示了对象的交互序列。这些可以使用UML顺序图或协作图来表示。顺序模型是动态模型。

3. 状态机模型, 展示了对象如何在事件响应中改变自己的状态。这些可以使用UML状态图来表示。状态机模型是动态模型。

子系统模型是一种有用的静态模型, 展示了如何将一个设计组织为逻辑上相关的对象组。图7-4中展示了这种类型的模型, 其中展示的是气象站系统中的子系统。除了子系统模型, 还可以设计详细的对象模型, 展示系统中的对象以及它们的关联关系(继承、泛化、聚集等)。然而, 进行太多的建模存在危险。不应当在此时做出关于实现的详细决策, 这些决策最好留到系统实现时再考虑。

顺序模型是动态模型, 描述了每个交互模式下所发生的对象交互序列。当描述一个设计时, 应当为每个有意义的交互产生一个序列模型。如果已经开发了一个用况模型, 那么每一个所识别出的用况都应该有一个顺序模型。

图7-7是一个顺序模型的例子, 表示为一个UML顺序图。这个图展示了当一个外部系统向气象站请求汇总数据时发生的交互序列。顺序图要从上到下阅读。

1. SatComms(气象站命令)对象接收来自气象信息系统的请求, 要求从一个气象站收集一个气象报告。它确认收到该请求。发送消息上的线形箭头表示外部系统不会等待回复, 而是可以继续其他处理。

2. SatComms对象通过一个卫星链路发送消息给WeatherStation(气象站)对象以创建一个所收集的气象数据的汇总。同样, 这里的线形箭头表示SatComms对象不用挂起自己以等待回复。

3. WeatherStation对象发送一个消息给Commslink(通信链路)对象以汇总气象数据。在这种情况下, 三角形箭头表示WeatherStation对象类的实例需要等待回复。

4. Commslink对象调用WeatherData(气象数据)对象的汇总方法并等待回复。

5. 气象数据汇总被计算出来, 并通过Commslink对象返回给WeatherStation对象。

6. WeatherStation对象接着调用SatComms对象, 将汇总后的数据通过卫星通信系统传送给气象信息系统。

SatComms和WeatherStation对象可以被实现为并发的进程, 它们的执行可以被挂起和恢复。SatComms对象实例监听来自外部系统的消息, 对这些消息进行解码, 并初始化气象站操作。

顺序图被用于对一组对象的组合行为进行建模, 但是你可能还想按照消息或事件对一个对象或子系统的行为进行总结。为此, 可以使用一个状态机模型来展示对象实例如何根据所收到消息改变状态。如第5章所述, UML包括描述状态机模型的状态图。

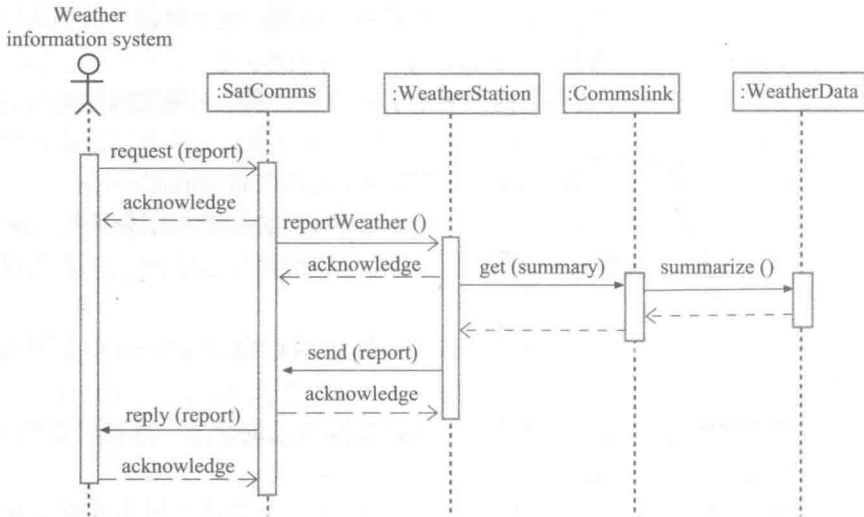


图 7-7 描述数据收集的顺序图

图 7-8 是一个气象站系统的状态图，其中展示了该系统如何响应各种不同的服务请求。

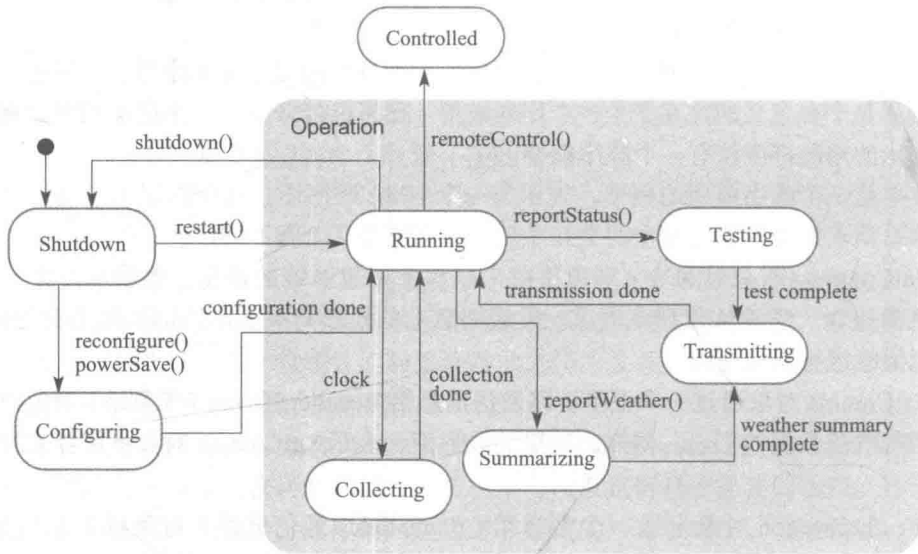


图 7-8 气象站状态图

可以按照以下方式阅读图 7-8。

1. 如果系统状态是 Shutdown（关闭），那么可以对 restart()、reconfigure() 或 powerSave() 消息进行响应。带有黑色圆点的没有标签的箭头表示 Shutdown 状态是初始状态。restart() 消息使得状态转换为正常运行状态。powerSave() 和 reconfigure() 消息都会使系统转换为对自身进行重配置的状态。此状态图显示，只有在系统已经关闭的情况下才允许进行重配置。

2. 在 Running（运行）状态下，系统期待进一步的消息。如果收到了 shutdown() 消息，那么对象返回关闭状态。

3. 如果收到 reportWeather() 消息，那么系统进入 Summarizing（汇总）状态。当汇总完

成后，系统进入 Transmitting（发送）状态，在此状态下信息被发送给远程系统。然后系统返回 Running 状态。

4. 如果收到一个来自时钟的信号，那么系统会进入 Collecting（收集）状态，在此状态下系统收集来自仪器的数据。每个仪器按顺序接受指令收集来自相关联的传感器的数据。

5. 如果收到 remoteControl() 消息，那么系统进入一个受控的状态，在此状态下系统对另一组来自远程控制室的消息进行应答。这些没有在图中显示。状态图是很有用的一种系统或对象操作的高层模型。

然而并不是系统中的所有的对象都需要一个状态图。一个系统中的许多系统对象都很简单，他们的操作可以很容易地在不使用状态模型的情况下进行描述。

7.1.5 接口规格说明

设计过程的一个重要部分是设计中所包含的构件之间的接口的规格说明。你需要刻画接口的规格说明，以使得对象和子系统并行进行设计。一旦确定了一个接口的规格说明，其他对象的开发者可以假设该接口将会被实现。

接口设计关注刻画一个对象或一组对象的接口的细节。这意味着要定义由一个对象或者一组对象所提供服务的型构（signature）和语义。接口在 UML 中可以用类图一样的表示法来刻画。然而，接口没有属性部分，而名称部分应当包含 UML 构造型（stereotype）«interface»。接口的语义可以使用对象约束语言（Object Constraint Language, OCL）来定义。第 16 章中将会讨论 OCL 的使用，其中会解释可以如何使用 OCL 来描述构件的语义。

接口设计不应当在一个接口设计中包含数据表示的细节，因为一个接口规格说明中并不会定义属性。然而，接口设计应当包含访问和更新数据的操作。由于数据表示被隐藏了，因此可以在不影响使用该数据的对象的情况下很容易地修改数据表示。这样产生的设计从内在上讲可以具有更好的可维护性。例如，一个堆栈的数组（array）表示可以在不影响使用堆栈的其他对象的情况下改为列表（list）表示。与之相对应的，一个对象模型中的属性通常都应当被暴露出来，因为这是描述对象的基本特性的一种最清楚的方式。

对象与接口之间并不是简单的一对一关系。同一对象可以有多个接口，其中每个接口都是一个对于该对象所提供的方法的视角。这一点在 Java 中可以直接得到支持，其中接口是独立于对象进行声明的，而对象会“实现”接口。同样，一组对象也可以通过单个接口一起访问到。

图 7-9 展示了为气象站定义的两个接口。左侧的接口是一个报告（Reporting）接口，定义了用于生成气象和状态报告的操作的名称。这些直接与 WeatherStation 对象中的操作相对应。远程控制（Remote Control）接口提供了 4 个操作，它们映射到 WeatherStation 对象的同一个方法上。在这个例子中，每个不同的操作都是在与 remoteControl 方法相关联的命令字符串中进行编码的，如图 7-6 所示。

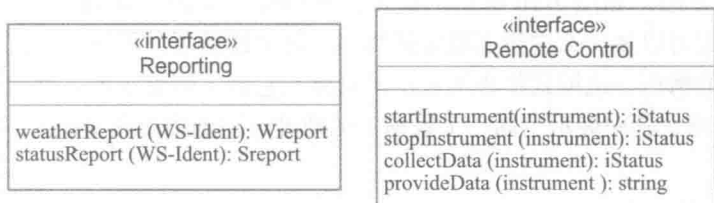


图 7-9 气象站接口

7.2 设计模式

设计模式是从 Christopher Alexander (Alexander 1979) 所提出思想中派生出来的, 他认为建筑设计中存在一定的共性模式, 这些模式令人喜欢并且有效。模式是一种对于问题及其解决方案的本质的描述, 从而使得解决方案可以在不同的环境中进行复用。模式不是一种详细的规格说明, 应当将模式理解为一种对于逐渐累积的智慧和经验的描述, 一种对某个共性问题的经过成功尝试的解决方案。

下面这段话引自 Hillside 小组的网站 (hillside.net/patterns/), 该网站专门维护关于模式的信息。其中描述了模式在复用中的角色:

模式和模式语言是描述最佳实践、好的设计的方式, 其中捕捉了相关的经验, 从而使其他人复用这些经验成为可能^①。

模式对于面向对象软件设计已经产生了巨大的影响。除了作为针对共性问题的经过验证的解决方案之外, 模式还已经成为一种谈论设计的词汇表。因此, 可以通过描述你所使用的模式来解释你的设计。最著名的例子是由 Gamma 等人 (被称为“四人帮”) 在他们 1995 年出版的关于模式的书 (Gamma et al. 1995) 中所描述的那些最著名的设计模式。其他重要的模式描述还包括由来自西门子 (一个大型的欧洲技术公司) 的作者所出版的一系列书 (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann, Henney, and Schmidt 2007a, 2007b)。

模式是一种复用其他设计者的知识和经验的方式。设计模式通常与面向对象设计相关。广泛发布的模式经常依赖于继承和多态等对象特性来提供通用性。然而, 在模式中封装经验这一通用原则可以同样适用于任何类型的软件设计。例如, 你可以拥有针对实例化可复用的应用系统的配置模式。

Gamma 等人在他们关于模式的书中定义了设计模式的 4 个基本元素。

1. 一个名字, 作为对模式的有意义的参照。
2. 一个问题域的描述, 解释了该模式何时适用。
3. 一个对于设计解决方案的各个部分、它们之间的关系以及它们的职责的描述。这并不是一个具体的设计描述, 这是一个设计解决方案的模板, 可以以不同的方式进行实例化。模板经常可以通过图形化的方式进行表达, 展示解决方案中的对象和对象类之间的关系。
4. 一个对效果的陈述——应用该模式的结果以及权衡。这有助于设计者理解一个模式是否可以用于某个特定的情形中。

Gamma 及其合著者将问题描述分解为动机 (对于模式为什么有用的一种描述) 和适用性 (对于该模式可以使用的情形的描述)。在解决方案的描述下, 他们刻画了模式结构、参与者、协作以及实现。

为了展示模式的描述, 这里使用 Gamma 等人设计模式的书中的观察者 (Observer) 模式作为例子进行介绍, 如图 7-10 所示。这个描述中包含了前面提到的 4 个基本描述元素, 同时对于这个模式可以做什么进行了简要的陈述。这个模型可以用于需要以不同的方式展示一个对象的状态的情形。该模式将必须展示的对象与它的不同展示形式分离开了。这一点在图 7-11 中有所体现, 其中展示了同一个数据集的两种不同的图形化呈现方式。

^① Hillside 小组: hillside.net/patterns

模式名称：观察者

描述：将一个对象的状态的呈现与对象本身分离开，允许为对象提供不同的呈现方式。当对象状态变化时，所有的呈现都会自动得到通知并进行更新以反映变化。

问题描述：在很多情况下你都必须为状态信息提供多种呈现方式，例如，一个图形化呈现方式和一个表格呈现方式。这些呈现方式在对信息进行规格说明时并不都是已知的。所有不同的呈现方式都应该支持交互，在状态发生变化时所有呈现都必须更新。

这个模式的使用情形是，状态信息需要不止一种呈现方式，并且维护状态信息的对象不需要知道所使用的特定的呈现格式。

解决方案描述：解决方案包括两个抽象对象 Subject（主题，即被观察者）和 Observer（观察者），以及两个继承了相关抽象对象属性的具体对象 ConcreteSubject（具体主题）和 ConcreteObserver（具体观察者）。抽象对象包括适用于所有情形的通用操作。需要呈现的状态在 ConcreteSubject 中进行维护，该对象继承了 Subject 的操作从而允许该对象增加和移除观察者（每个观察者对应一种呈现方式）以及在状态发生变化时发出通知。

ConcreteObserver 维护了 ConcreteSubject 状态的一份拷贝，并且实现了 Observer 的 Update（）接口，该接口允许这些拷贝能够被同步更新。ConcreteObserver 自动呈现状态并在任何时候当状态更新时自动反映变化。

该模式的 UML 模型如图 7-12 所示。

效果：主题对象只知道抽象的观察者对象，而不知道具体类的细节。因此，这些对象之间的耦合被最小化了。由于缺少这些知识，提高呈现性能的优化可能无法实现。对主题对象的变化可能导致生成与之相关的针对观察者的一组更新，而其中一些可能并没有必要。

图 7-10 观察者模式

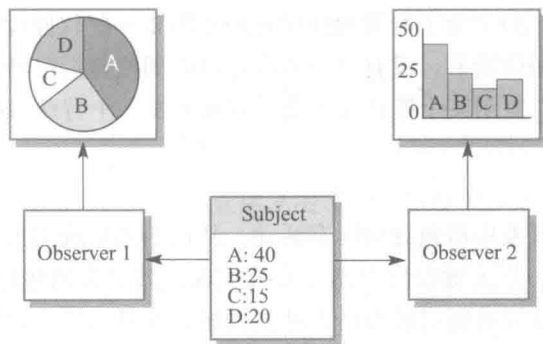


图 7-11 多种呈现方式

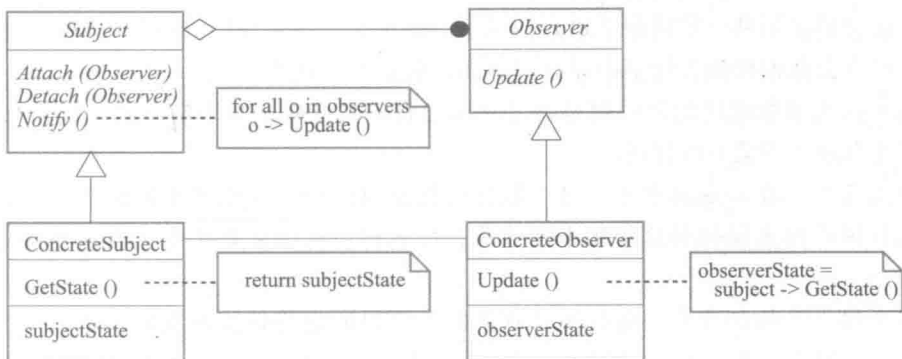


图 7-12 观察者模式的 UML 模型

为了在设计中使用模式，你需要认识到你所面对的任何设计问题都可能会有一个与之相关联、可以应用的模式。Gamma 等人关于设计模式的书中描述了以下这些关于这种问题的例子。

1. 向多个对象告知一些其他对象的状态发生了变化（观察者模式）。
2. 整理面向一些相关的且经常是增量地开发出来的对象的接口（门面模式）。
3. 为一个合集（collection）中的元素提供一种标准的访问方式，不用考虑合集是如何实现的（迭代器模式）。
4. 允许在运行时扩展一个已有的类的功能（装饰者模式）。

模式支持高层的概念复用。当试图复用可执行的构件时，不可避免地会受到这些构件的实现中所做出的详细设计决策的约束和限制。这些决策范围很广，从已经用于实现这些构件的特定算法到构件接口中的对象和类型。当这些设计决策与需求相冲突时，可能就无法复用该构件了，或者会在复用中引入问题。使用模式意味着对思想进行复用，但可以对实现进行调整以适应正在开发的系统。

当开始设计一个系统时，很难提前知道是否会需要某个特定的模式。因此，在设计过程中使用模式经常要经过开发设计、体验问题、然后认识到可以使用某个模式的过程。如果关注最初那本设计模式的书中所描述的 23 个通用模式，那么这当然是可能的。然而，如果所遇到的问题与这些模式都不一样，那么要从已经提出来的成百上千个不同的模式中找到一个合适的模式，会感到有些困难。

模式是伟大的思想，但是为了有效地使用它们还需要一些软件设计经验。你必须能够认识到一个模式可以被应用的情形。没有经验的程序员，即使他们已经读过模式的书，也总是会发现很难决定是否可以复用某个模式或者是否需要开发一个特殊目的的解决方案。

7.3 实现问题

软件工程包括软件开发中所包含的所有活动，从初始的系统需求直到部署后的系统维护和管理。这个过程的一个关键阶段当然是系统实现，其中会创建软件的一个可执行版本。实现可能会包括：使用高层或底层编程语言开发程序，或者对通用的成品系统进行裁剪和适配以满足一个组织的特定需求。

这里假设本书的大多数读者都能够理解编程原则，并且已经有一些编程经验。由于本章的目的是提供一种与语言无关的方法，因此这里并不会关注与良好的编程实践相关的一些问题，因为这会要求用到一些特定于语言的例子。这里将介绍实现中的一些对软件工程尤其重要并且经常不会在编程相关的书中介绍的方面，包括如下这些。

1. 复用。大多数现代的软件都是通过复用已有的构件或系统来构造的。在开发软件时，应该尽可能多地利用已有的代码。
2. 配置管理。在开发过程中，每个软件构件都可能会创建很多不同的版本。如果不用一个配置管理系统来保持对这些版本的追踪，那么很可能在系统中包含这些构件的错误版本。
3. 宿主机-目标机开发。运行生产软件的计算机通常与软件开发环境中的计算机并不相同。软件一般都是在—台计算机上开发（宿主机）；而在另一台计算机上运行（目标机）。宿主机和目标机有时候是同一种类型，但却经常完全不同。

7.3.1 复用

从 20 世纪 60 年代到 20 世纪 90 年代,大多数新软件都是从头开始开发的,一般都是通过使用某种高级编程语言编写所有代码。唯一显著的软件复用是对编程语言库中的功能和对象的复用。然而,成本和进度压力使这一方法越来越不可行,特别是对于商业系统和基于互联网的系统。因此,基于复用已有软件的开发方法现在已经成为许多不同类型系统开发的基本准则。基于复用的方法现在已经在很多领域得到广泛使用,包括所有类型的基于 Web 的系统、科学软件,以及越来越多的嵌入式系统软件。

软件复用可以在多个不同的级别上发生,如图 7-13 所示。

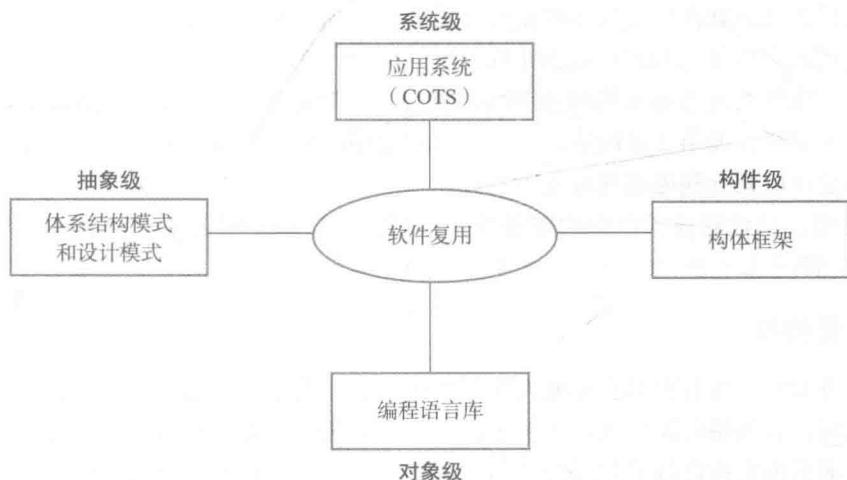


图 7-13 软件复用

1. 抽象级。在这个级别上,并不是直接复用软件而是在软件的设计中使用关于一些成功的抽象的知识。设计模式和体系结构模式(见第6章)都是面向复用的抽象知识表示的方式。

2. 对象级。在这个级别上,直接复用来自库中的对象而不是自己写代码。为了实现这种类型的复用,必须找到合适的库,并且确定其中的对象和方法是否提供了所需要的功能。例如,如果需要在 Java 程序中处理电子邮件消息,那么可以使用 JavaMail 库中的对象和方法。

3. 构件级。构件是对象和对象类的合集,它们一起运行以提供相关的功能和服务。你经常都要通过增加一些你自己的代码来对构件进行适配和扩展。一个构件级复用的例子是利用一个框架来构建自己的用户界面。这种框架包含一组实现事件处理、显示管理等通用对象类。需要增加与显示数据之间的连接并且写代码来定义特定的显示细节(例如屏幕布局和颜色)。

4. 系统级。在这个级别上,复用整个应用系统。此过程中通常会包括对这些系统的某种方式的配置。这可以通过增加和修改代码(如果你在复用一个软件产品线)来实现,或者通过使用系统自身的配置接口来实现。大多数商业化系统现在都是用这种方式构建的,其中会对通用的应用系统进行适配和复用。有时候这种方法还可能会包括对多个应用系统的集成来创建一个新系统。

通过复用已有的软件,可以更快地开发新系统,而且风险和成本更低。由于被复用的软

件已经在其他应用中得到了验证，它们应该会比新软件更可靠。然而，还是会有一些与复用相关的成本。

1. 寻找可复用的软件以及评价其是否满足需要所花费的时间成本。你可能还必须对软件进行测试以确保它可以在你的环境中工作，特别是该环境与软件的开发环境不一样的时候。

2. 在适用的情况下，购买可复用软件的成本。对于大型的成品系统，这些成本可能会很高。

3. 适配和配置可复用软件构件或系统以反映正在开发的系统的需求的成本。

4. 可复用软件元素相互之间集成（如果你在使用不同来源的软件）以及与所开发的新代码相集成的成本。集成来自不同提供者的可复用软件可能会很难并且代价较高，因为这些提供者针对他们各自的软件将会如何被复用会做出一些相互冲突的假设。

如何复用已有的知识和软件应当是在开始一个软件开发项目时要考虑的第一件事情。应当在详细设计软件之前考虑复用的可能性，因为你可能要对自己的设计进行调整以适应对已有软件资产的复用。如第2章所述，在一个面向复用的开发过程中，要搜索可复用元素然后修改需求和设计以充分利用这些可复用元素。

由于复用在现代软件工程中的重要性，本书的第三部分用了多个章节来介绍这一主题（第15、16、18章）。

7.3.2 配置管理

在软件开发中，变化总是在不断发生，因此变更管理绝对是很重要的。当多个人参与软件系统开发时，必须确保团队成员不会干扰其他人的工作。也就是说，如果两个人都在开发一个构件，那么他们的修改必须进行协调，否则一个程序员所做的修改可能会覆盖其他人的工作。此外，还要确保每个人都可以访问到软件构件的最新版本，否则开发人员可能要重做已经做过的工作。当一个系统的新版本出现问题时，必须能够回到该系统或构件此前的一个可以工作的版本上。

顾名思义，配置管理是管理一个不断变化的软件系统的一般过程。配置管理的目的是支持系统集成过程以使所有的开发者都可以以一种受控的方式访问项目的代码和文档，找出代码和文档做了哪些修改，以及对构件进行编译和链接来创建一个系统。如图7-14所示，有以下4个基本的配置管理活动。

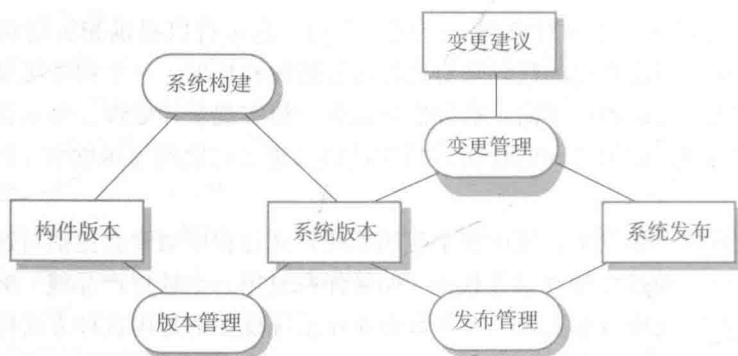


图 7-14 配置管理

1. 版本管理，为保持对软件构件的不同版本的追踪提供支持。版本管理系统包括一些

手段来协调多个程序员之间的开发。这些方法会阻止一个开发人员覆盖别人提交到系统中的代码。

2. 系统集成, 为帮助开发人员定义用于创建每一个系统版本的构件的版本提供支持。然后, 这一描述被用于通过编译和链接所需要的构件来自动构建一个系统。

3. 问题追踪, 为用户报告 bug 和其他问题提供支持, 为所有开发人员查看谁在处理这些问题以及问题何时被修复提供支持。

4. 发布管理, 会向客户发布一个软件系统的新版本。发布管理关注规划新发布的功能并为了分发而对软件进行组织。

软件配置工具支持以上每一个活动。这些工具通常安装在一个集成开发环境(例如 Eclipse)中。版本管理可以使用一个版本管理系统来支持, 例如 Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) 或者 Git (Loeliger and McCullough 2012), 它们可以支持多地点、多团队的开发。系统集成支持可以被构建到语言里面或者依赖于一个独立的工具集(例如 GNU 的构建系统)。bug 追踪 (bug tracking) 或问题追踪 (issue tracking) 系统(例如 Bugzilla) 用于报告 bug 和其他问题, 并且对它们是否被修复保持追踪。围绕 Git 系统构建的一个很全的工具集列表可以在 Github (<http://github.com>) 上找到。

由于变更管理和配置管理在专业化的软件工程中的重要性, 本书在第 25 章对它们进行更加详细的介绍。

7.3.3 宿主机 - 目标机开发

大多数专业化的软件开发都是基于宿主机 - 目标机模型的(图 7-15)。软件在一台计算机(宿主机)上开发, 但是在另一台机器(目标机)上运行; 更泛化一些的话, 可以说一个开发平台(宿主机)和一个执行平台(目标机)。一个平台不仅仅是硬件, 还包括所安装的操作系统和其他支持性的软件(例如数据库管理系统); 或者对于开发平台而言, 一个交互式的开发环境。

有时候, 开发平台和执行平台是一样的, 这使软件的开发和测试可以在同一台机器上进行。因此, 如果用 Java 开发, 那么目标环境是 Java 虚拟机。原则上讲, 这个目标环境在每台计算机上都一样, 因此程序应该可以从一台机器迁移到另一台机器上。然而, 特别是对于嵌入式系统和移动系统, 开发和执行平台是不一样的, 需要将所开发的软件移动到执行平台上进行测试, 或者在开发机上运行一个模拟器。

模拟器经常在开发嵌入式系统时使用, 对硬件设备(例如传感器)以及系统将被部署的环境中的事件进行模拟。模拟器加快了嵌入式系统的开发过程, 因为每个开发人员都可以有自己的执行平台, 而不需要将软件下载到目标硬件上。然而, 模拟器的开发很昂贵, 因此可用的模拟器通常都是那些针对最流行的硬件体系结构的。

如果目标系统已经安装了需要使用的中间件或其他软件, 那么要能够使用这些软件来测试系统。有时候在开发机上安装这些软件是不现实的, 即使开发机和目标平台是一样的, 也

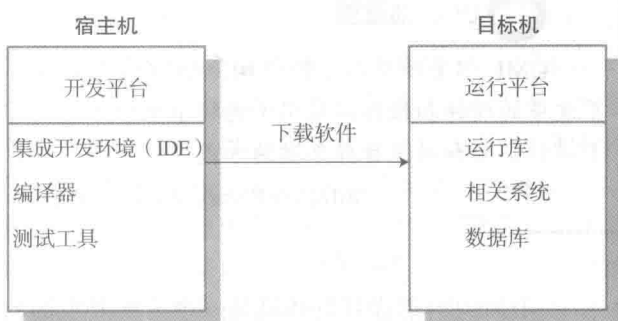


图 7-15 宿主机 - 目标机开发

可能因为许可证的限制而无法实现这一目的。如果是这种情况，那么需要将所开发的代码转移到执行平台上，从而对系统进行测试。

软件开发平台应当提供一系列工具来支持软件工程过程。这些可能会包括：

1. 一个允许开发人员创建、编辑和编译代码的集成编译器以及语法制导的编辑系统；
2. 一个语言调试系统；
3. 图形化编辑工具，例如编辑 UML 模型的工具；
4. 可以自动在程序的新版本上运行一组测试的测试工具，例如 JUnit；
5. 支持重构和程序可视化的工具；
6. 管理源代码版本以及集成和构建系统的配置管理工具。

除了这些标准化的工具，开发系统还可以提供更加专业的工具，例如静态分析器（在第 12 章中介绍）。通常，面向团队的开发环境还会包括一个运行变更和配置管理系统的共享服务器，还可能有一个支持需求管理的系统。

软件开发工具现在通常都会安装在一个集成开发环境中（Integrated Development Environment, IDE）。一个集成开发环境是一组位于一些公共框架和用户界面中的支持软件开发的软件工具。一般而言，集成开发环境都是支持使用特定编程语言（例如 Java）的开发。针对特定语言的集成开发环境可以是特别开发的，也可以在通用集成开发环境基础上加入特定的语言支持工具。



UML 部署图

UML 部署图显示了软件构件如何物理地部署在处理器上。也就是说，部署图显示了系统中的硬件和软件以及用于连接系统中不同构件的中间件。从本质上可以认为，部署图是一种定义和描述目标环境的方式。

<http://software-engineering-book.com/web/deployment/>

一个通用的集成开发环境是一个为所开发的软件提供数据管理设施以及允许各种工具一起工作的集成机制的框架，可以容纳各种软件工具。最广为人知的通用集成开发环境是 Eclipse 环境（<http://www.eclipse.org>）。这个环境基于一个插件体系结构，从而使其可以面向不同的语言（例如 Java）和应用领域进行定制。因此，你可以安装 Eclipse，并且通过增加插件来按照自己的特定目的对其进行裁剪。例如，可以增加一组插件以支持使用 Java 的网络化系统开发（Vogel 2013），或者使用 C 语言进行嵌入式系统工程。

作为开发过程的一部分，需要针对所开发的软件将如何在目标平台上部署做出决策。这对于嵌入式系统是很直观的，其中目标机通常是单台计算机。然而，对于分布式系统而言，需要决定相关构件部署在哪些特定的平台上。在做出这一决策时必须考虑如下问题。

1. 一个构件的硬件和软件需求。如果一个构件是为一个特定的硬件体系结构而设计的，或者依赖于其他一些软件系统，那么很明显该构件要被部署到提供所需要的硬件和软件支持的平台上。
2. 系统的可用性需求。高可用性的系统可能要求将构件部署到多个平台上，这意味着如果发生平台失效，那么该构件的其他实现是可用的。

3. 构件通信。如果有许多构件间通信,那么最好将这些构件部署在同一个平台上或者部署在物理上相互接近的多个平台上。这样做降低了通信延迟,即一个消息由一个构件发出到另一个构件收到消息之间的延迟。

可以使用 UML 部署图来描述你对于硬件和软件部署的决策,其中可以显示软件构件如何分布在不同的硬件平台之上。

如果你正在开发一个嵌入式系统,你可能不得不考虑目标机的特性,例如,它的物理大小、电量、对于传感器事件的实时响应要求、效用器的物理特性以及它的实时操作系统。本书将在第 21 章介绍嵌入式系统工程。

7.4 开源开发

开源开发是一种软件开发方法,其中软件系统的源代码被公开并邀请志愿者参加开发过程(Raymond 2001)。开源开发源于自由软件基金会(Free Software Foundation, www.fsf.org),他们主张源代码不应该是私有的,而应该总是面向用户开放,使他们可以按照自己的意愿检查并修改代码。其中有一个假设,代码将由一个小的核心小组而不是代码的用户进行控制和开发。

开源软件通过使用互联网来招募更大规模的志愿开发人员扩展了这一思想。他们中的许多人本身也是代码的用户。至少在原则上,一个开源项目的任何贡献者都可以报告和修复 bug,并提出新的特征和功能建议。然而,在实践中,成功的开源系统仍然依赖于一个核心开发人员小组来控制软件的变更。

开源软件是互联网和软件工程的支柱。Linux 操作系统是最广泛使用的服务器系统,开源的 Apache web 服务器也是一样。其他重要并且得到广泛使用的开源产品还包括 Java、Eclipse 集成开发环境、MySQL 数据库管理系统等。Android 操作系统安装在成百上千万的移动设备上。主要的计算机产业巨头,例如 IBM 和 Oracle,都支持开源运动并且将他们的软件建立在开源产品基础上。此外,还有成千上万的其他没那么知名的开源系统和构件可以使用。

获取开源软件通常很便宜甚至是免费的,你通常可以免费下载开源软件。然而,如果想获得文档和相应的支持,那么可能必须要为此付费,但是成本通常相当低。使用开源产品的另一个关键的优势是广泛使用的开源系统非常稳定。这些系统有大量用户自愿去修复软件中的问题,而不是把这些问题报告给开发者并等待系统新的发布版本。bug 的发现和修复通常要比私有软件要快。

对于参与软件开发的公司而言,有两个关于开源的问题是必须要考虑的。

1. 正在开发的产品应该利用开源构件吗?
2. 应该使用开源的方法来开发自己的软件吗?

对于这些问题的回答取决于所开发的软件的类型以及开发团队的背景和经验。如果正在开发一个用于销售的软件产品,那么上市时间和降低成本是很重要的。如果正在一个存在可用的高质量开源系统的领域中开发软件,那么可以通过使用这些系统来节省时间和费用。然而,如果你正在针对一组特定的组织需求开发软件,那么使用开源构件可能不是一个可行的选项。你可能必须要将你的软件和已有的与可用的开源系统不兼容的系统相集成。然而,即使这样,修改开源系统也比重新开发所需要的功能更快和更便宜。

许多软件产品公司现在都在使用开源方法来开发,特别是针对专业的系统。他们的业务模型并不依赖于销售软件产品,而是提供对于该产品的服务。他们认为将开源社区纳入进来

会使软件开发更快同时更加便宜,并且将为该软件创建一个用户社区。

一些公司认为采用开源方法会将机密的业务知识透露给他们的竞争对手,因此不愿意采用这一开发模型。然而,如果你在一个小公司工作并将自己的软件开源,那么你可以请客户放心,因为如果你的公司歇业了,那么他们可以自己支持该软件。

发布一个系统的源代码并不意味着来自更大范围内的社区的人们将要帮助进行开发。大多数成功的开源产品都是平台产品而不是应用系统。可能对专业的应用系统感兴趣的开发者数量很有限。让一个软件系统开源并不能够保证社区的参与度。Sourceforge 和 GitHub 上有成千上万的开源项目都只有很少的下载次数。然而,如果你的软件用户对该软件未来是否可用存在疑虑,那么让软件开源就意味着用户可以获取自己的拷贝,并且放心他们不会失去对代码的访问权。

7.4.1 开源许可证

虽然开源软件开发的一个基本原则是,源代码应当是免费提供的,但是这并不意味着任何人都可以随心所欲地处置这些代码。从法律意义上讲,代码的开发者(一个公司或一个人)拥有代码。他们可以通过在一个开源软件许可证中包含法律约束条件来对代码如何使用加以限制(St. Laurent 2004)。一些开源开发者认为如果一个开源构件被用于开发一个新系统,那么该系统也应当是开源的。其他人则愿意让别人在没有这些限制的情况下使用他们的代码。所开发的系统可以是私有的并作为闭源系统出售。

大多数开源许可证(Chapman 2010)都是以下3个通用模型中某一个的变体。

1. 自由软件基金会(GNU)通用公共许可证(General Public License, GPL)。这是一个所谓的互惠许可证,可以简单理解为,如果你使用 GPL 许可证下的开源软件,那么你必须让你的软件开源。

2. 自由软件基金会(GNU)宽通用公共许可证(Lesser General Public License, LGPL)。这是 GPL 许可证的一个变体,按照该许可证你可以编写链接到开源代码的构件而不用发布这些构件的源代码。但是,如果你修改了受许可证保护的构件,那么你必须将其发布作为开源软件。

3. 伯克利标准分发(Berkley Standard Distribution, BSD)许可证。这是一个单向的许可证,意味着你没有责任重新发布任何对于开源代码所做出的修改。你可以在所销售的私有系统中包含这些代码。如果你使用了开源构件,你必须承认这些代码最初的创造者。MIT 许可证是 BSD 许可证的一个变体,条款相似。

许可证问题很重要,因为如果你在一个软件产品中使用开源软件,那么你可能会被强制要求服从许可证条款来使你自己的产品开源。如果你试图销售自己的软件,你可能希望对其保密。这意味着你可能会希望避免在开发中使用 GPL 许可证的开源软件。

如果你在构造运行在开源平台上的软件,但是并没有复用开源构件,那么许可证不是个问题。然而,如果你将开源软件嵌入你的软件中,那么你需要建立相应的过程和数据库来保持对已经使用的开源代码及其许可证条件的追踪。Bayersdorfer (Bayersdorfer 2007) 建议管理使用了开源代码的项目的公司应当做到以下几点。

1. 建立一个系统来维护关于所下载和使用的开源构件的信息。你必须针对每一个构件保存一个在使用构件时有有效的许可证的副本。许可证可能会发生变化,因此你要知道你已认可的那些许可证条款。

2. 了解不同类型的许可证,并理解一个构件在使用前是如何确定许可证的。你可能会决定在某个系统中使用一个构件而不在另一个系统中使用它,因为你计划以不同的方式使用这

些系统。

3. 了解构件的演化路径。你需要知道一点关于开发构件的开源项目的事情以理解这些构件未来会如何变化。

4. 进行关于开源软件的教育。仅仅建立规程来确保符合许可证条款是不够的,还需要对开发人员进行关于开源软件以及开源许可证的教育。

5. 建立审计系统。处于严格的交付期限压力下的开发人员可能会试图违反许可证条款。如果可能的话,应当使用软件来侦测违反许可证条款的情况并进行阻止。

6. 参与开源社区。如果你依赖于开源产品,那么你应该参与相关的社区并帮助支持他们的开发。

开源开发方法是几种软件商业模型之一。在这种模型中,公司发布他们软件的源代码并且提供收费的附加服务以及与此相关的建议。他们还可以销售基于云的软件服务,对于那些不具有管理自己的开源系统以及系统的专业版本的用户来说,这是一种很有吸引力的选项。因此,开源很有希望在开发和分发软件方面发挥越来越大的作用。

要点

- 软件设计和实现是相互交织的活动。设计中的详细程度取决于所开发的系统的类型以及所使用的是计划驱动的方法还是敏捷方法。
- 面向对象设计过程所包括的活动有设计系统体系结构、识别系统中的对象、使用不同的对象模型描述设计、对构件接口进行文档化描述。
- 在一个面向对象设计过程中会产生一系列不同的模型,其中包括静态模型(类模型、泛化模型、关联模型)和动态模型(顺序模型、状态机模型)。
- 构件接口必须精确地进行定义,以使得其他对象可以使用这些接口。UML 接口的结构可以被用来定义接口。
- 当开发软件时,应该总是考虑复用已有软件的可能性,可以是复用构件、服务或完整的系统。
- 配置管理是对一个不断演化的软件系统的修改过程进行管理。当一个团队中的成员相互协作开发软件时,配置管理是十分重要的。
- 大多数软件开发都是宿主机-目标机开发。在宿主机上使用一个 IDE 来开发软件,然后将软件转移到一台目标机上执行。
- 开源开发很重要的一点是对外公开源代码。这意味着许多人都可以对软件的修改和改进提出建议。

阅读推荐

《Design Patterns: Elements of Reusable Object-oriented Software》是最初的软件模式手册,其中面向大范围的读者介绍了软件模式。(E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison-Wesley, 1995)

《Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd ed》这本书中 Larman 清晰地介绍了面向对象设计,同时也介绍了 UML 的使用。这本书对在设计过程中如何使用模式进行了很好的介绍。虽然这本书已经问世十多年了,但是在同类型的书中仍然是最好的一本。(C. Larman, Prentice-Hall, 2004)

《Producing Open Source Software: How to Run a Successful Free Software Project》这本书对开源软件的背景、许可证问题、运行一个开源开发项目的可行性进行了全面的介绍。(K. Fogel, O'Reilly Media Inc., 2008)

关于软件复用的推荐阅读在第 15 章中, 关于配置管理的推荐阅读在第 25 章中。

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap7/>

支持视频的链接: <http://software-engineering-book.com/videos/implementation-and-evolution/>

关于气象信息系统的更多信息:

<http://software-engineering-book.com/case-studies/wilderness-weather-station/>

练习

- 7.1 使用图 7-3 中所示的表格化表示法对气象站用况“报告状态和重配置”进行描述。你应当对这里所需要的功能做出合理的假设。
- 7.2 假设 Mentcare 系统在使用面向对象方法进行开发。画一个显示至少 6 个该系统可能的用况的用况图。
- 7.3 使用 UML 对于对象类的表示法设计下列对象类, 识别属性和操作。根据你自己的经验来决定与这些对象相关联的属性和操作。
 - 一个移动电话或者平板电脑上的消息通信系统;
 - 一个个人计算机的打印机;
 - 一个个人音乐系统;
 - 一个银行账户;
 - 一个图书馆目录。
- 7.4 使用图 7-6 中所识别的气象站对象作为起点, 识别该系统中可能会用到的其他对象。为你所识别出的对象设计一个继承层次。
- 7.5 开发一个气象站的设计来展示数据收集子系统和收集气象数据的仪器之间的交互。使用顺序图来展示这些交互。
- 7.6 识别下列系统中可能的对象并为它们开发一个面向对象的设计。在考虑设计时, 你可以做出关于这些系统的任何合理假设。
 - 一个小组日程和时间管理系统, 希望能够支持对一组同事之间的会议和预约进行时间安排。当做出包含多人的预约时, 系统要找出他们的日程中一个共同的时间段, 并且将预约安排到这个时间段上。如果找不到共同的时间段, 系统与用户进行交互以重新安排他的个人日程, 从而使预约可以找到合适的时间。
 - 一个完全自动化运营的加油站即将开业。驾驶员通过一个与油泵相连的读卡器刷他的信用卡; 所刷的卡通过与一个信用卡公司的计算机通信进行验证, 并建立一个燃料限量。接着驾驶员就可以获得所需要的燃料了。当燃料供应完成后, 油泵软管回归皮套。信用卡在扣费后返回卡。如果信用卡不合法, 那么油泵在加油之前就会返回卡。
- 7.7 画一个顺序图展示小组日程系统在一组人正在安排一个会议时对象间的交互。
- 7.8 画一个 UML 状态图来展示小组日程系统或者加油系统可能的状态变化。
- 7.9 举例解释为什么一组人在开发一个软件产品时配置管理系统很重要。

- 7.10 一个小公司开发了一个可以专门为每个客户专门配置的专业软件产品。新客户通常都有一些特定的需求要加入到系统中，他们会为这些需求的开发和集成支付费用。该软件公司有一个机会去投标一个新项目，这将会使客户基数翻倍。新客户希望参与一些系统的配置。解释为什么在这种情况下让这个软件产品开源对于拥有该软件的公司而言可能是个好主意。

参考文献

- Abbott, R. 1983. "Program Design by Informal English Descriptions." *Comm. ACM* 26 (11): 882–894. doi:10.1145/182.358441.
- Alexander, C. 1979. *A Timeless Way of Building*. Oxford, UK: Oxford University Press.
- Bayersdorfer, M. 2007. "Managing a Project with Open Source Components." *ACM Interactions* 14 (6): 33–34. doi:10.1145/1300655.1300677.
- Beck, K., and W. Cunningham. 1989. "A Laboratory for Teaching Object-Oriented Thinking." In *Proc. OOPSLA'89 (Conference on Object-Oriented Programming, Systems, Languages and Applications)*, 1–6. ACM Press. doi:10.1145/74878.74879.
- Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.
- . 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.
- Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.
- Chapman, C. 2010. "A Short Guide to Open-Source and Similar Licences." *Smashing Magazine*. <http://www.smashingmagazine.com/2010/03/24/a-short-guide-to-open-source-and-similar-licenses/>
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.
- Loeliger, J., and M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly & Associates.
- Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly & Associates.
- Raymond, E. S. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly & Associates.
- Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.
- St. Laurent, A. 2004. *Understanding Open Source and Free Software Licensing*. Sebastopol, CA: O'Reilly & Associates.
- Vogel, L. 2013. *Eclipse IDE: A Tutorial*. Hamburg, Germany: Vogella GmbH.
- Wirfs-Brock, R., B. Wilkerson, and L. Weiner. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall.

软件测试

目标

本章的目标是介绍软件测试和软件测试过程。阅读完本章后，你将：

- 理解测试的各个阶段，从开发过程中的测试到系统客户所做的验收测试；
- 了解可以帮助你选择有利于发现程序缺陷的测试用例的技术；
- 理解测试先行的开发，其中会在编写代码前设计测试并且自动运行这些测试；
- 了解3种截然不同的测试，即构件测试、系统测试、发布测试；
- 理解开发测试和用户测试的区别。

测试的目的是显示一个程序做了希望做的事情以及在程序投入使用之前发现其中的缺陷。当测试软件时，会使用人造的数据执行程序。要对测试运行的结果进行检查以发现错误、异常现象，或者关于程序的非功能性属性的信息。

测试软件时，你在试图做以下两件事情。

1. 向开发人员和客户展示软件满足其需求。对于定制化软件，这意味着针对需求文档中的每一个需求都应该有至少一个测试。对于通用的软件产品，这意味着所有将被包含在产品发布中的系统特征都要有测试。你还可以测试特征的组合以检查是否存在不希望出现的特征交互。

2. 找出可能导致软件行为不正确、出现不希望的行为，或行为不符合规格说明的输入或输入序列。这些是由于软件中的缺陷（bug）导致的。当你测试软件以发现缺陷时，你将试图根除不希望看到的系统行为，例如系统崩溃、不希望看到的与其他系统的交互、不正确的计算，以及数据损坏等。

上面的第一点是确认测试，你会使用一组反映系统的期望使用方式的测试用例来测试系统是否正确运行。第二点是缺陷测试，设计测试用例来暴露缺陷。缺陷测试中的测试用例可以故意有一些不清楚的地方，并且不需要反映软件在正常情况下是如何使用的。当然，这两种测试方法并没有明确的边界。在确认测试过程中，会发现系统中的缺陷；在缺陷测试过程中，一些测试会显示程序满足其需求。

图 8-1 显示了确认测试和缺陷测试之间的区别。将被测试的系统想象成一个黑盒。系统从输入集合 I 那里接收输入，生成一个输出集合 O 中的输出。输出中有一些会是错误的。这些错误的输出集合 O_e 是系统对输入集合 I_e 中的输入的响应而生成的。缺陷测试中优先考虑的是

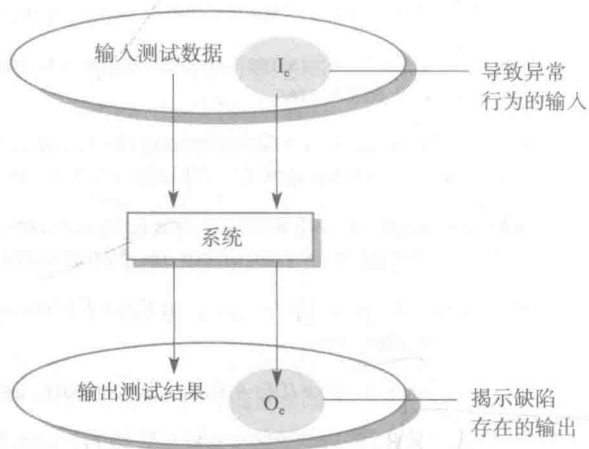


图 8-1 程序测试的一个输入-输出模型

找到 I_e 中的输入, 因为这些输入揭示了系统中的问题。确认测试包含使用 I_e 之外的正确输入的测试。这些输入激励系统生成所期望的正确输出。

测试无法显示软件没有缺陷或者软件在每种情形下的行为都与规格说明一致。你所忽视的某一个测试总是有可能进一步发现系统中新的问题。就如软件工程发展的早期贡献者 Edsger Dijkstra 所说 (Dijkstra 1972):

“测试只能显示错误的存在, 而不是显示没有错误存在^①”

测试是更广阔的软件验证和确认 (Verification and Validation, V & V) 过程的一部分。验证和确认并不相同, 虽然它们经常被混淆。软件工程先驱之一 Barry Boehm 简洁地将二者之间的区别表达为 (Boehm 1979):

- 确认 (validation): 我们在构造正确的产品吗?
- 验证 (verification): 我们在以正确的方式构造产品吗?

验证和确认过程关注检查所开发的软件是否满足其规格说明, 并且是否提供了为软件开发付费的人所期望的功能。这些检查过程在有了需求之后马上就开始了, 并且一直持续并贯穿整个软件开发过程的所有阶段。

软件验证是检查软件是否满足其所声明的功能性和非功能性需求的过程。软件确认是一个更加泛化的过程。确认的目的是确保软件满足客户的期望。确认的含义不仅仅是检查与规格说明的符合性以展示软件做了客户期望它做的事情。确认十分重要, 因为如本书第4章中所述, 需求陈述并不总是反映系统客户和用户的真实愿望或需要。

验证和确认过程的目的是建立软件系统“符合目的”的信心。这意味着系统必须能够足够好地满足所期望的使用方式。所需要的信心水平取决于系统的目的、系统用户的期望, 以及针对该系统的当前市场环境。

1. 软件目的。软件越关键, 它的可靠性越重要。例如, 相比一个用于原型化新的产品设想的演示系统所需的信息水平, 用于控制一个安全关键性系统的软件所需的信心水平要高得多。

2. 用户期望。由于此前对于问题频现、不可靠的软件的经历, 用户有时对软件质量的期望并不高。当他们的软件失效时, 他们并不会奇怪。当一个新系统安装后, 用户可能会容忍一些失效, 因为使用系统的收益大于失效恢复的开销。然而, 随着一个软件产品的位置逐渐巩固, 用户会期望它变得越来越可靠。因此, 可能会需要对系统此后的版本进行更加彻底的测试。

3. 市场环境。一个软件公司在将一个系统推向市场时必须考虑竞争产品、客户愿意为系统支付的价格, 以及系统交付所需要的日程。在一个竞争性的环境中, 公司可能会决定在充分测试和调试之前发布一个程序, 因为这样可以最先进入市场。如果一个软件产品或应用非常便宜, 用户可能会愿意容忍较低的可靠性水平。

除了软件测试, 验证和确认过程还可以包括软件审查 (inspection) 和评审 (review)。审查和评审分析并检查系统需求、设计模型、程序源代码, 甚至所建议的系统测试。这些都属于“静态的”验证和确认技术, 其中不需要执行软件来进行验证。图 8-2 显示了软件审查和测试在软件过程的不同阶段支持验证和确认。其中的箭头表示这些技术可以使用的过程阶段。

① Dijkstra, E. W. 1972. “The Humble Programmer.” Comm. ACM 15 (10): 859-66. doi:10.1145/355604.361591

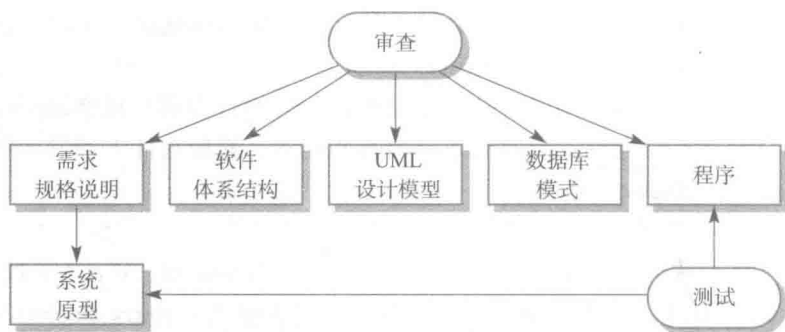


图 8-2 审查和测试

审查主要关注系统的源代码，但其他任何可读的软件表示（例如，需求或设计模型）也可以进行审查。当你审查一个系统时，你要利用关于系统、其应用领域、编程或建模语言的知识来发现错误。

与测试相比软件审查有如下 3 个优势。

1. 在测试过程中，一个错误可能会掩盖（隐藏）其他错误。当一个错误导致非预期的输出时，你永远无法确定此后的输出异常是由于新的错误导致的，还是原来那个错误的副作用导致的。由于审查并不会运行系统，你并不需要担心错误之间的相互影响。因此，一次审查会议可能会发现一个系统中的很多错误。

2. 一个系统的不完整版本可以在不需要额外成本的情况下进行审查。如果一个程序是不完整的，那么需要开发特殊的测试装置来测试已有的部分。显然，这将增加系统的开发成本。

3. 除了查找程序中的缺陷之外，审查还可以考虑范围更广阔的程序质量属性，例如，与标准的符合性、可移植性、可维护性等。通过审查可以寻找效率不高、不合适的算法，以及可能导致系统难以维护和更新的不好的编程风格。

程序审查是一个比较老的思想，一些研究和实验已经表明审查在缺陷发现方面比程序测试更有效。Fagan (Fagan 1976) 在报告中提到，程序中超过 60% 的错误可以使用非正式的程序审查来发现。据称在净室过程 (Prowell et al. 1999) 中，程序审查可以发现超过 90% 的缺陷。

然而，审查无法取代软件测试。审查不善于发现由于程序的不同部分之间的非预期交互、时间性问题或者系统性能上的问题而导致的缺陷。在小的公司或开发小组中，设立一个独立的审查团队可能很困难并且很昂贵，因为所有潜在的团队成员可能都是软件的开发者。

第 24 章（质量管理）中将更详细地介绍评审和审查。第 12 章将介绍静态分析，对一个程序的源代码进行自动分析以发现异常。在本章中，将关注测试和测试过程。

图 8-3 是一个计划驱动的开发所使用的关于传统测试过程的抽象模型。测试用例是关于测试的输入以及期望的系统输出（测试结果）的规格说明，加上一个关于所测试的是什么陈述。测试数据是已经设想好的用于测试一个系统的输入。测试数据有时可以自动生成，但是测试用例的自动生成是不可能的。理解假设让系统做什么的人必须参与定义所期望的测试结果。然而，测试执行可以被自动化。测试结果可以自动与期望的结果进行比较，因此不需要人来寻找测试运行中的错误和异常。

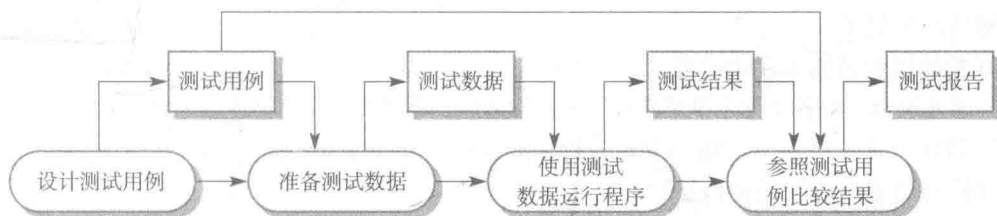


图 8-3 软件测试过程模型

通常，一个商业化软件系统必须经历以下 3 个测试阶段。

1. 开发测试，系统在开发过程中进行测试以发现 bug 和缺陷。系统设计者和程序员一般都会参与这个测试过程。

2. 发布测试，一个独立的测试团队在系统发布给用户之前测试系统的完整版本。发布测试的目的是检查系统是否满足系统利益相关者的需求。

3. 用户测试，一个系统的用户或潜在的用户在他们自己的环境中测试系统。对于软件产品，“用户”可以是一个决定软件是否可以推向市场、发布和销售的内部市场组。验收测试是一种用户测试，其中客户正式地测试系统以决定是否应该从系统供应商那里接收该系统，或者系统是否需要进一步的开发。



测试计划

测试计划关注测试过程中所有活动的进度和资源安排。其中包括定义测试过程，考虑可用的人和资源。通常都会创建一个测试计划，其中定义了测试什么、所预测的测试进度、测试将如何记录等。对于关键性系统，测试计划还可以包括要在软件上运行的测试的详细信息。

<http://software-engineering-book.com/web/test-planning/>

在实践中，测试过程通常会混合手工测试和自动化测试。在手工测试中，测试人员使用一些测试数据运行程序，并将运行结果与他们的期望相比较，然后记录所发现的不符之处并报告给程序开发者。在自动化测试中，测试被编码在程序中，在所开发的系统每次测试时运行这些测试程序。这种方式比手工测试要快，特别是当测试中包括回归测试（重新运行前面的测试以确定对程序的修改没有引入新的 bug）时。

不幸的是，测试永远无法做到完全自动化，因为自动化测试只能确定一个程序做了人们认为它应该做的事情。在实践中不可能使用自动化测试来测试系统看起来的样子（例如一个图形用户界面），或者测试一个程序意料之外的副作用。

8.1 开发测试

开发测试包括由开发系统的团队所执行的所有测试活动。软件的测试者通常是开发该软件的程序员。一些开发过程使用程序员 / 测试人员对 (Cusamano and Selby 1998)，其中每个程序员都有一个相关联的测试人员来开发测试并协助进行测试过程。对于关键性系统可能要使用一个更加正式的过程，其开发团队中有独立的测试小组，该小组负责开发测试并维护详

细的测试结果记录。

开发测试包括如下 3 个阶段。

1. 单元测试，对各个程序单元或对象类进行测试。单元测试应当关注测试对象或方法的功能。
2. 构件测试，对多个不同的单元进行集成以创建一个复合构件。构件测试应当关注测试提供对构件功能的访问的构件接口。
3. 系统测试，对一个系统中的一些或全部构件进行集成并将系统作为一个整体进行测试。系统测试应当关注测试构件交互。

开发测试主要是一个缺陷测试过程，其中测试的目的是发现软件中的 bug。因此，开发测试通常和调试（在代码中定位问题并修改程序以修复问题的过程）交织在一起。



调试

调试是修复测试所发现的错误和问题的过程。基于来自程序测试的信息，调试人员使用他们的编程语言知识以及测试的期望输出来定位并修复程序错误。当你在调试一个程序时，你通常会使用交互式的工具来提供关于程序运行的额外信息。

<http://software-engineering-book.com/web/debugging/>

8.1.1 单元测试

单元测试是测试程序构件（例如，方法或对象类）的过程。各个函数或方法是构件的最简单的形式。测试应当用不同的输入参数来调用这些程序。可以使用 8.1.2 节中所介绍的测试用例设计方法来设计函数或方法测试。

在测试对象类时，应当设计测试来实现对该对象所有特征的覆盖。这意味着应当测试与该对象相关的所有操作，设置并且检查与该对象相关的所有属性的值，并且将该对象置于所有可能的状态之中。这意味着应当模拟导致状态变更的所有事件。

例如，考虑第 7 章中所介绍的例子中的气象站对象。该对象的属性和操作如图 8-4 所示。这个对象只有一个属性，即它的标识符。这个属性是一个常量，在气象站安装时进行设置。因此，只需要一个测试来检查该属性是否进行了适当的设置。要为与该对象相关联的所有方法（例如 `reportWeather` 和 `reportStatus`）定义测试用例。理想情况下，应当独立测试各个方法，但是在有些情况下也有必要对方法序列进行测试。例如，为了测试关闭气象站仪器的方法（`shutdown`），你需要先执行 `restart` 方法。

泛化或继承使得对象类的测试更加复杂。无法简单地在定义一个操作的类中对其进行测试，然后假设这个方法可以在继承该操作的所有子类中都能按照预期进行工作。被继承的操作可能会对于其他操作和属性做出一些假设。这些假设在一些继承该操作的子类中可能并不成立。因此，必须在使用所继承的操作的每一个地方对其进行测试。

为了测试气象站的状态，可以使用一个如第 7 章中所介绍的状态模型（见图 7-8）。使用这种模型，可以识别必须测试的状态转换

WeatherStation
identifier
reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

图 8-4 气象站对象接口

序列并定义驱使这些状态转换的事件序列。从原则上讲,应当测试每种可能的状态转换序列,虽然在实践中这可能会过于昂贵。气象站系统中应当测试的状态序列的一些例子如下:

Shutdown → Running → Shutdown

Configuring → Running → Testing → Transmitting → Running

Running → Collecting → Running → Summarizing → Transmitting → Running

只要有可能都应当尽量自动化单元测试。在自动化单元测试中,可以使用一个测试自动化框架,例如 JUnit (Tahchiev et al. 2010) 来编写并运行程序测试。单元测试框架提供了可以扩展用来创建特定测试用例的通用测试类。它们可以运行你所实现的所有测试,并且报告测试是否成功(经常通过一些图形化用户界面)。整个测试套件经常可以在几秒钟内运行完,因此有可能在你每次修改程序时都执行所有测试。

一个自动化测试包括以下 3 个部分。

1. 一个设置部分,其中会按照当前测试用例(即输入和期望输出)对系统进行初始化。
2. 一个调用部分,其中会调用要进行测试的对象或方法。
3. 一个断言部分,其中会将调用的结果与所期望的结果进行比较。如果断言结果是真,那么测试是成功的;否则测试失败。

有时候,测试的对象会依赖其他还没有实现的对象,或者依赖会导致测试过程很慢的对象。例如,如果一个对象调用一个数据库,那么就需要在使用数据库之前进行一个很慢的设置过程。在这种情况下,你可能会决定使用 Mock 对象。

Mock 对象与所模拟的外部对象具有同样的接口。例如,一个模拟数据库的 Mock 对象可能只有一些组织为数组的数据项。访问这些 Mock 对象很快,不存在调用数据库和访问磁盘带来的额外开销。与之相似,Mock 对象可以被用于模拟异常操作或稀有事件。例如,如果系统要在一天之中的一些特定时间上采取动作,那么 Mock 对象可以简单地返回那些时间,而不用管实际的时钟时间。

8.1.2 选择单元测试用例

测试很昂贵并且很耗时间,因此选择有效的单元测试用例是很重要的。这里的“有效”意味着两件事情。

1. 测试用例应当显示出,当按照期望的方式使用时正在测试的构件做了期望它做的事情;

2. 如果构件中存在缺陷,那么测试用例可以揭示这些缺陷。

因此,应当设计两类测试用例。第一类测试用例应该反映程序的正常运行,并且显示出构件可以工作。例如,如果你在测试一个创建并初始化一条新的病人记录的构件,那么你的测试用例应当显示出所添加的记录在数据库中存在,并且它的字段已经按照所指定的值进行了设置。另一类测试用例应当基于常见的问题通常会在哪里出现的测试经验。这些测试用例应当使用异常的输入来确定这些输入得到了适当的处理,并且不会使构件崩溃。

以下是两种可以帮助你选择测试用例的策略。

1. 划分测试,识别出各个具有共同特性并且应当以同样的方式处理的输入分组。你应当从这些分组中的每一个中都选择测试用例。

2. 基于指南的测试,使用测试指南来选择测试用例。这些指南反映了程序员在开发构件时经常会犯的各种错误的经验。

一个程序的输入数据和输出结果可以看作是是具有共同特性的集合成员。这些集合的例子包括整数、负数、菜单选择。程序通常对一个集合中的所有成员都表现出近似的行为。也就是说,如果你测试一个进行计算的程序并且需要两个正数,那么你会期望程序对于所有的正数都表现出相同的行为。

由于这种等价行为,这些类有时候被称为“等价划分”(equivalence partition)或“等价域”(Bezier 1990)。一种系统性的测试用例设计方法就是建立在面向一个系统或构件识别所有输入和输出划分的基础上。测试用例设计就可以从这些划分中选取输入或输出。划分测试既可以用于为系统设计测试用例,又可以用于为构件设计测试用例。

在图 8-5 中,左侧的大阴影椭圆表示所测试的程序所有可能的输入集合。较小的不带阴影的椭圆表示等价划分。一个被测程序应当以同样的方式对一个输入等价划分中的所有成员进行处理。

输出等价划分是其中所有输出划分都有一些共性的成分。有时候输入和输出等价划分之间存在一对一映射。然而并不总是如此,有时候可能需要单独定义输入等价划分,其中输入唯一的共性特征是它们所产生的输出在同一个输出划分之中。图 8-5 中左边椭圆中的阴影区域表示非法输入。右边椭圆中的阴影区域表示可能发生的异常,即对非法输入的响应。

一旦已经识别出了一组划分,就可以从每一个划分中选择测试用例。一个关于测试用例选取的有效经验法则是,选取划分边界上的测试用例,再加上靠近划分中点的测试用例。其原因是设计者和程序员在开发一个系统时倾向于考虑典型的输入值。这些值可以通过选取划分的中点来进行测试。边界值经常是非典型的(例如,针对输入值 0 的行为可能与其他非负数字的行为不一样,因此有时候会被开发人员忽视。程序失效经常会在处理这些非典型值时发生。

可以通过使用程序规格说明或用户文档,并且根据预测哪种类型的输入值最有可能发现错误的经验来识别划分。例如,假设有一个程序规格说明明确程序接受 4~10 个大于 10 000 的五位整数作为输入。根据这些信息可以识别输入划分以及可能的测试输入值,如图 8-6 所示。

当使用一个系统的规格说明来识别等价划分时,可以称为“黑盒测试”。你不需要了解任何关于系

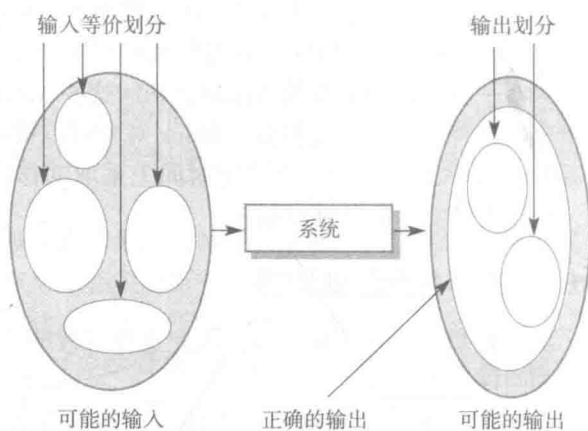


图 8-5 等价划分

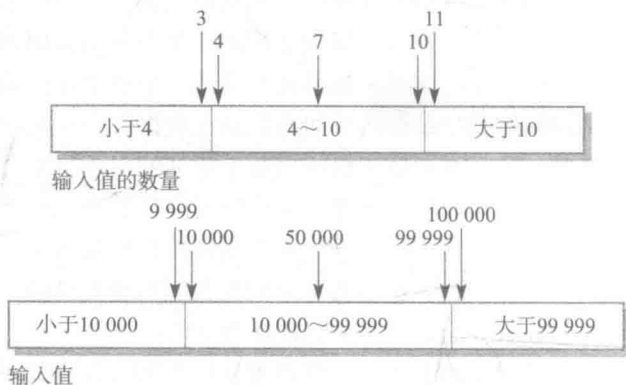


图 8-6 等价划分

统如何工作的知识。有时候需要通过“白盒测试”来对黑盒测试进行补充。白盒测试中会考虑程序的代码来找到其他可能的测试。例如，代码可能会包含处理不正确输入的异常。那么可以利用这一知识来识别“异常划分”，即应当采用同样的异常处理的那些取值范围。



路径测试

路径测试是一种测试策略，其目的是运行贯穿一个构件或程序的每一个独立的执行路径。如果每一个独立路径都被执行，那么构件中所有的语句一定都被执行了至少一次。所有的条件语句的真和假两种情况也都进行了测试。在一个面向对象开发过程中，路径测试可以用来测试与对象相关的方法。

<http://software-engineering-book.com/web/path-testing/>

等价划分是一种有效的测试方法，因为它有助于说明程序员在处理划分边缘上的输入时经常犯的错误。也可以使用测试指南来帮助选择测试用例。指南封装了关于何种类型的测试用例能够有效发现错误的知识。例如，当你在测试带有序列、数组或列表的程序时，可以帮助发现缺陷的指南通常包括以下几项。

1. 使用只有单个值的序列来测试软件。程序员经常很自然地认为序列由多个值组成，并且有时会在他们的程序中体现这一假设。因此，如果碰到只有单个值的序列，那么程序可能会出错。

2. 在不同的测试中使用多个大小不同的序列。这降低了具有缺陷的程序由于输入的一些偶然特性而碰巧产生正确输出的可能性。

3. 考虑让测试访问到序列中最前面、中间、最后面的元素。这种方法可以揭示划分边界上的问题。

Whittaker 的书（Whittaker 2009）中包括了很多可以用于测试用例设计的指南的例子。其中所建议的一些最通用的指南包括：

- 选择迫使系统生成所有错误消息的输入；
- 设计导致输入缓冲溢出的输入；
- 多次重复同样的输入或输入序列；
- 驱使生成非法的输出；
- 驱使计算结果太大或太小。

随着你不断获得测试经验，你可以开发你自己的选择有效测试用例的指南。下一节中将给出更多的测试指南的例子。

8.1.3 构件测试

软件构件经常由多个相互交互的对象组成。例如，在气象站系统中，重配置构件包括处理重配置的各个方面的对象，可以通过构件接口（见第7章）访问这些对象的功能。因此，测试复合构件应当关注显示构件接口的行为与其规格说明相一致。可以假设针对构件中的各个对象的单元测试都已经完成了。

图 8-7 描述了构件接口测试的想法。假设构件 A、B 和 C 已经被集成到一起以创建一个更大的构件或子系统。测试用例并不是应用于单个构件而是通过组合这些构件形成的复合构件的接口。复合构件中的接口错误可能无法通过测试各个对象来发现,因为这些错误来自于构件中对象之间的交互。

程序构件之间存在不同类型的接口,因此可能发生的接口错误有如下这些类型。

1. 参数接口。这种接口中数据或者有时候函数引用从一个构件传递到另一个。一个对象中的方法有一个参数接口。

2. 共享存储接口。这种接口中会在构件之间共享一块存储。数据由一个子系统放在存储中,然后由其他子系统从那里读取数据。这种接口在嵌入式系统中使用,其中的传感器会创建由其他系统构件读取和处理的数据。

3. 过程式接口。这种接口中一个构件封装了一组可以由其他构件调用的过程。对象和可复用构件有这种形式的接口。

4. 消息传递接口。这种接口中一个构件通过传递消息来从另一个构件那里请求服务。返回消息包括执行服务的结果。一些面向对象系统有这种形式的接口,客户-服务器系统也是这样。

接口错误是复杂系统中最常见的一种错误形式之一 (Lutz 1993)。这些错误包括以下 3 种类型。

- 接口误用。一个调用构件在调用其他构件并且在使用其他构件的接口时犯了一个错误。这类错误在参数接口中很常见,其中参数可能类型错误、传递顺序不对或者传递的参数数量不对。
- 接口误解。一个调用构件误解了所调用构件的接口的规格说明,并对它的行为做出了假设。所调用的构件行为与预期不符,从而导致调用构件中的非预期行为。例如,一个二分查找方法在被调用时所传入的参数可能是一个未排序的数组。这样的查找会失败。
- 时间性错误。这些错误在使用共享存储或消息传递接口的实时系统中发生。数据的生产者和消费者可能以不同的速度运行。除非在接口设计中进行了特殊考虑,否则消费者可能会访问到过时的信息,因为信息的生产者没有更新共享接口的信息。

测试接口缺陷很困难,因为一些接口缺陷只会是一些非常条件下才会表现出来。例如,一个对象将一个队列实现为一个固定长度的数据结构,一个调用对象可能会假设该队列被实现为一个无穷数据结构,因此在输入一个数据项时没有检查队列溢出。

只能通过设计一系列测试用例来迫使队列溢出,这样才能发现上述情况。这些测试应当检查调用对象如何处理溢出。然而,由于这种情况很稀少,测试人员可能会在为队列对象编写测试集时觉得不值得检查。

不同模块或对象中的缺陷之间的交互有可能会产生进一步的问题。一个对象中的缺陷可能只在一些其他对象按照非预期的方式表现时才会被发现。例如,一个对象调用其他对象以接受一些服务,调用对象假设返回结果是正确的。如果所调用的服务有某种错误,那么所返

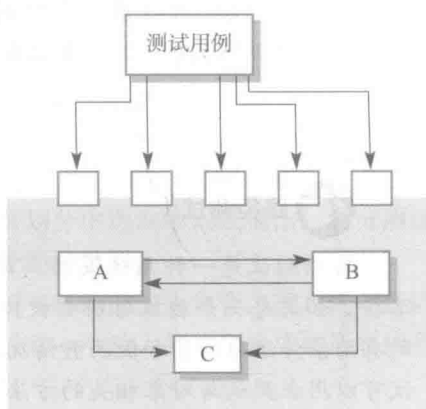


图 8-7 接口测试

回的值可能是合法的但是不正确。因此,该问题并不能马上发现,而只能当进行后续的一些使用该返回值的计算出错时才被发现。

接口测试的一些通用指南包括以下几条。

1. 检查要测试的代码,识别每一个对于外部构件的调用。设计一组测试,其中对于外部构件的参数值处于它们取值范围的极限边界上,这些极限值最有可能揭示接口不一致性。

2. 对于任何在接口上传递的指针,总是使用空指针参数测试接口。

3. 对于任何通过过程接口调用的构件,设计故意导致构件失效的测试。不同的失效假设是规格说明误解最常见的一种情况。

4. 在消息传递系统中使用压力测试。这意味着你应当设计一些测试,会产生比实践中正常情况下可能出现的多得多的消息。这是一种发现时间性错误的有效方法。

5. 当多个构件通过共享存储交互时,设计改变这些构件被激活的顺序的测试。这些测试可能会发现程序员对于共享数据产生和消费的顺序的隐式假设。

有时候使用审查和评审而不是测试来寻找接口错误会更好。审查可以关注构件接口以及在审查过程中问到的关于所假设的接口行为的问题。

8.1.4 系统测试

开发过程中的系统测试包括集成构件以创建一个系统版本,然后测试集成后的系统。系统测试检查构件是否兼容、是否能正确交互,以及是否能在正确的时间跨越接口传输正确的数据。系统测试显然与构件测试存在重叠,但是二者存在以下两个重要的区别。

1. 在系统测试过程中,独立开发的可复用构件和成品系统可能会与新开发的构件相集成。这样就可以对完整的系统进行测试了。

2. 由不同团队成员或子团队开发的构件可以在这个阶段进行集成。系统测试是一个集体的过程而不是个别的过程。在有些企业中,会由一个独立的没有参加过该系统设计和实现的测试团队来进行系统测试。

所有系统都有一些涌现性的行为。这意味着一些系统功能和特性只有当把构件放到一起时才会显现出来。这些行为可能是计划好的涌现性行为,这种行为必须进行测试。例如,将一个身份认证构件和一个更新系统数据库的构件相集成,这样就有了一个限制只允许授权用户进行信息更新的系统特性。然而,有时候涌现性行为不是计划好的并且是不希望看到的。必须开发测试来确认系统只做那些希望它做的事情。

系统测试应当关注测试构成一个系统的构件和对象之间的交互,还可能会在可复用构件或系统与新构件集成时对它们进行测试,以确认它们是否如预期的一样运作。这一集成测试应当发现那些只有当一个构件在一个系统中被其他构件使用时才会发现的构件 bug。交互测试还有助于发现构件开发者对系统中的其他构件所产生的误解。

由于用况关注交互,因此基于用况的测试是一种有效的系统测试方法。系统中的每一个用况通常都是由多个构件或对象实现的。测试用例会驱动这些交互发生。如果开发了一个顺序图来建模用况实现,那么可以看到参与交互的对象或构件。

在野外气象站的例子中,系统软件向一个远程的计算机报告汇总后的气象数据,如图 7-3 所示。图 8-8 显示了气象站系统在响应为地图系统收集数据的请求时的操作序列。可以用这幅图来识别将要进行测试的操作,并帮助设计测试用例来执行这些测试。因此,发出一个气象数据报告请求可以引发下面这个方法线程的执行:

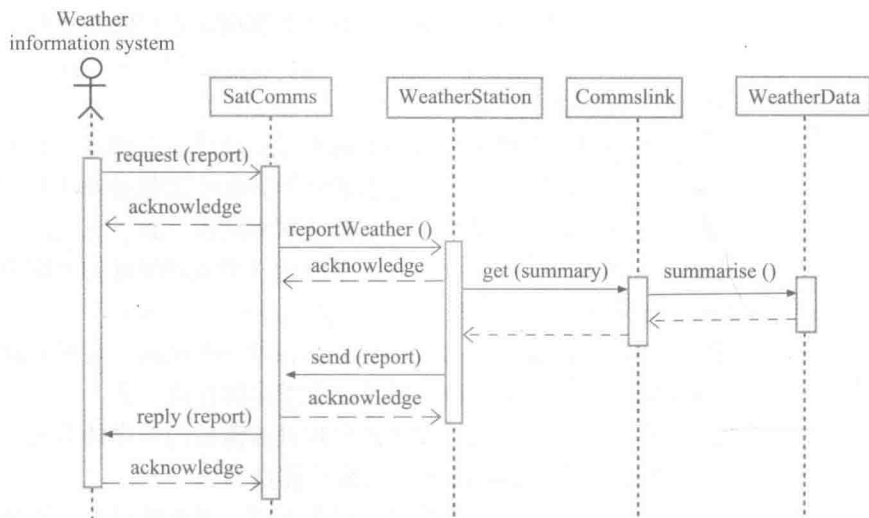


图 8-8 收集气象数据顺序图

SatComms:request → WeatherStation:reportWeather → Commslink:Get (summary)
→ WeatherData:summarize

顺序图帮助设计人员设计出所需要的特定的测试用例，因为它显示了需要什么样的输入以及产生什么样的输出，例如这个例子中的如下这些信息。

1. 获取报告的请求的输入应当有一个相关的应答。一个报告应当最终从请求中返回。在测试过程中，应当创建可以用于确认报告的组织是否正确的汇总数据。

2. 一个针对 WeatherStation 的报告请求的输入会导致生成一个汇总报告。通过创建对应于你为 SatComms 的测试所准备的汇总信息原始数据，并且检查 WeatherStation 对象是否正确地产生了这一汇总，可以独立对其进行测试。这些原始数据还被用于测试 WeatherData 对象。

当然，图 8-8 中的顺序图中进行了简化，从而让其不显示异常。一个完整的用况 / 场景测试必须考虑这些异常并且确保它们都得到了正确处理。对于大多数系统，想知道有多少系统测试是重要的以及何时应该停止测试都是很难的。彻底的测试，即对每一个可能的程序执行序列都进行测试，是不可能的。因此，测试必须基于所有可能的测试用例的一个子集。理想情况下，软件公司应当有如何挑选这一子集的策略。这些策略可能会基于通用的测试策略，例如所有的程序语句都应该被测试至少一次；或者也可以基于系统使用的经验并关注测试运行系统的特征。例如：

1. 通过菜单访问的所有系统功能都应该进行测试。
2. 通过同样的菜单访问的功能组合（例如文本格式化）必须进行测试。
3. 在任何提供了用户输入的地方，所有的功能必须同时使用正确的和不正确的输入进行测试。

来自于一些主要的软件产品（例如，字处理软件或电子表格）的经验来看，产品测试过程中通常会使用相似的指南。当软件的特征分开独立使用时，它们通常都可以工作。但是，如同 Whittake 所述（Whittaker 2009），当没那么常用的特征的组合没有被一起测试过的时候，经常会出问题。他给出了一个例子，在一个广泛使用的字处理软件中，使用带有多列

布局的脚注导致了不正确的文本布局。

系统测试的自动化通常比单元测试或构件测试要难。自动化的单元测试依赖于预测输出然后将这些预测编码到程序中。这一预测接下来会与结果进行比较。然而，一个系统实现所产生的输出则会很大或者很难预测。也许可以在不需要提前进行预测的情况下检查一个输出并确定它的可信度。



增量的集成和测试

系统测试包括集成不同的构件然后对所创建的集成后的系统进行测试。应当总是使用一种增量的方法来进行集成和测试，其中要集成一个构件，测试系统，集成另一个构件，再进行测试，等等。如果发生问题，那么问题很可能是由于与最近集成进来的构件的交互导致的。

增量的集成和测试对于敏捷方法很重要，会在每次集成一个新的增量之后执行回归测试。

<http://software-engineering-book.com/web/integration/>

8.2 测试驱动的开发

测试驱动的开发（Test-driven development, TDD）是一种软件开发方法，其中将测试和代码开发交织在一起（Beck 2002; Jeffries and Melnik 2007）。可以增量地开发代码以及开发一组针对该增量的测试。在已经开发好的代码通过所有它的测试之前，不会开始开发下一个增量。测试驱动的开发是作为 XP（极限编程）敏捷开发方法的一部分被引入的。然而，测试驱动的开发现在已经得到了主流的认同，既可以用于敏捷过程又可以用于基于计划的过程。

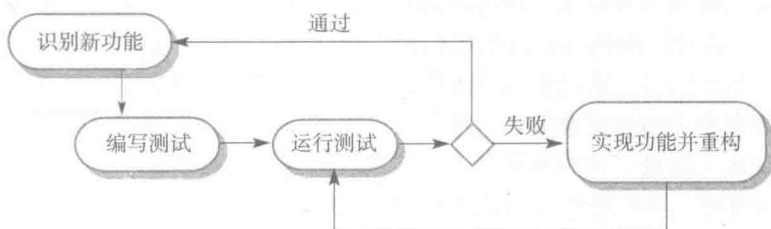


图 8-9 测试驱动的开发

基本的测试驱动的开发过程如图 8-9 所示，该过程中的步骤如下。

1. 首先识别所需要的功能增量。这通常应当比较小，并且可以通过不多的代码行就可以实现。
2. 为这个功能编写一个测试，并且将其实现为一个自动化测试。这意味着该测试可以执行并且会报告测试通过与否。
3. 接下来运行该测试以及所有其他已经实现的测试。起初还没有实现该功能，因此新的测试会失败。这是故意而为的，因为这表明该测试向测试集中增加了一些新东西。

4. 然后实现该功能并重新运行测试。其中还可能会包含重构已有的代码以对其进行改进, 并且向已有的实现中增加新的代码。

5. 一旦所有测试运行都成功了, 就可以转而实现下一个功能了。

一个自动化测试环境, 例如支持 Java 程序测试的 JUnit 环境 (Tahchiev et al. 2010), 对于测试驱动的开发很重要。由于代码是以很小的增量来开发的, 因此必须能够在每次增加新功能或者重构程序时运行每一个测试。因此, 测试嵌入在一个独立的程序中, 其中会运行测试并且调用所测试的系统。使用这种方法, 可以在几秒之内运行成百上千个不同的测试。

测试驱动的开发帮助程序员阐明他们关于一个代码段实际上应该做什么的想法。为了编写一个测试, 需要理解代码要做什么, 因为这一理解会使编写所需要的代码变得更容易。当然, 如果只有不完整知识或理解, 那么测试驱动的开发可能没什么帮助。

如果没有了解足够的信息以编写测试, 那么就无法开发所需要的代码。例如, 如果计算包含除法, 那么应当确保没有用 0 去除其他数字。如果忘记为此编写测试了, 那么相应的检查代码也不会出现在程序中。

除了更好的问题理解, 测试驱动的开发的其他好处还包括以下这些。

1. 代码覆盖。原则上讲, 所编写的每个代码段都应当有至少一个相关的测试。这样, 就可以有把握地相信系统中的所有代码都被执行过了。代码在编写时就会被测试, 因此可以在开发过程中的早期发现缺陷。

2. 回归测试。测试集随着程序的开发增量进行开发。可以总是运行回归测试来确认对程序的修改没有引入新的 bug。

3. 简化的调试。当一个测试失败时, 问题出在哪里应该很明显。新写的代码需要进行检查和修改。不需要使用调试工具来定位问题。关于测试驱动开发的使用报告建议说, 在测试驱动的开发中一般不需要使用自动化的调试器 (Martin 2007)。

4. 系统文档化。测试自身可以作为某种形式的文档来描述代码应该做什么。阅读测试可以使理解代码变得更容易。

测试驱动的开发的一个最重要的好处是降低了回归测试的开销。回归测试要在对一个系统进行了修改之后运行已经成功执行过的测试集合。回归测试要确保这些修改没有向系统中引入新的 bug, 并且新的代码与已有的代码按照预期的方式进行交互。当一个系统采用手工测试时, 回归测试很昂贵, 有时候不切实际, 因为所需要的时间和工作量开销非常高。因此不得不试图选择最相关的测试来重新运行, 而这样很容易遗漏重要的测试。

自动化测试极大降低了回归测试的开销。已有的测试可以快速、便宜地重新运行。在测试先行的开发中修改一个系统之后, 所有已有的测试都必须在增加任何进一步的功能之前成功运行。作为一个程序员, 你可以有信心确保你所增加的新功能没有导致已有的代码出现问题。

测试驱动的开发在开发新的软件时最有价值, 此时功能要么在新代码中实现、要么使用来自标准库中的构件来实现。如果复用大的代码构件或遗留系统, 那么需要作为一个整体为这些系统编写测试, 因为很难将它们分解为不同的可测试元素。增量的测试驱动的开发也不现实了。测试驱动的开发对于多线程系统效果可能也不好。不同的线程在不同的测试运行中可能会在不同的时间交织在一起, 因此会产生不同的结果。

如果使用测试驱动的开发, 那么仍然需要一个系统测试过程来验证系统, 也就是说确保系统满足所有系统利益相关者的需求。系统测试也会测试性能、可靠性, 并且确保系统没有做不应该做的事情, 例如产生不希望的输出。Andrea (Andrea 2007) 对于如何对测试工具

进行扩展从而将系统测试的某些方面与测试驱动的开发相集成给出了一些建议。

测试驱动的开发现在已经是一种广泛使用的主流软件测试方法。大多数采用这一方法的程序员都对其感到满意，并且发现它是一种更有效的软件开发方式。此外，也有人宣称使用测试驱动的开发可以鼓励更好的程序结构以及更好的代码质量。然而，针对这些说法的验证实验还没有什么明确的结果。

8.3 发布测试

发布测试是指对一个系统在开发团队以外进行使用的一个特定发布进行测试的过程。通常，系统发布是面向客户和用户的。然而，在一个复杂的项目中，发布也可能会面向开发相关系统的其他团队。对于软件产品而言，发布可能会面向后续会准备对其进行销售的产品管理人员。

在开发过程中，发布测试和系统测试有以下两个重要的区别。

1. 系统开发团队不应该负责发布测试。
2. 发布测试是一个确认检查的过程，其目的是确保一个系统满足它的需求，并且运行得足够好可以由系统客户进行使用。由开发团队进行的系统测试应当关注发现系统中的 bug(缺陷测试)。

发布测试过程的主要目的是让系统的供应商确信系统足够好可以使用了。如果是这样，那么产品就可以发布或交付给客户了。因此，发布测试必须表明系统提供了所要求的功能、性能以及可依赖性，并且系统不会在正常使用过程中失效。

发布测试通常是一个黑盒测试过程，其中测试是基于系统规格说明产生的。系统被视为一个黑盒，它的行为只能通过研究它的输入和相关的输出来确定。这一测试的另一个名字是功能测试，这么叫是因为测试人员只关注功能而不是软件的实现。

8.3.1 基于需求的测试

好的需求工程实践的一个一般原则是需求应该是可测试的。也就是说，需求的编写应当使得可以为其编写测试，这样测试人员就可以确认需求是否满足。因此，基于需求的测试是一种系统性的测试用例设计方法，其中需要考虑每一个需求并为其创建一组测试。基于需求的测试是一种确认而非缺陷测试——试图展示系统已经适当地实现了它的需求。

例如，下面是 Mentcare 系统关注检查药物过敏的需求：

如果已知一个病人对某些特定的药物过敏，那么开该药物的处方必须导致系统向用户发出一个警告消息；

如果一个开药者选择忽略一个过敏警告，那么他或者她必须提供一个理由说明为什么忽略这个警告。

为了检查这些需求是否得到了满足，可能需要开发以下这几个相关的测试。

1. 设置一条不包含已知的过敏症的病人记录。针对已知存在的过敏症开药方。检查确保系统不会发出警告消息。
2. 设置一条包含已知的过敏症的病人记录。针对该病人过敏的药物开处方，检查确保系统会发出警告消息。
3. 设置一条记录了两个或多个药物过敏的病人记录。分开对这些药物开药方，检查确保针对每个药物的警告都能正确发出。
4. 在药方中同时包括病人过敏的两种药物。检查确保两条警告都能正确地发出。

5. 开一个会发出过敏警告的药方并且忽略该警告。检查确保系统要求用户提供信息解释为什么忽略该警告。

从这个列表中可以看出，测试一个需求并不意味着只写一条测试就够了。通常不得不编写多个测试来确保已经很好地覆盖了该需求。还应当保持基于需求的测试的追踪关系记录，该追踪关系将测试链接到所测试的特定需求上。

8.3.2 场景测试

场景测试是一种发布测试方法，即设想出典型的使用场景并使用这些场景来为系统开发测试用例。一个场景是一个描述了系统可能的使用方式的故事。场景应当是现实的，真实的系统用户应当能够与其建立联系。如果在需求工程过程（见第4章）中使用过场景或用户故事，那么也许能够将它们复用作为测试场景。

在一个关于场景测试的短文中，Kaner（Kaner 2003）提到场景测试应当是一个具有可信性并且有一定复杂性的叙述性的故事。它应当触发利益相关者的积极性，也就是说利益相关者应当与该场景建立联系并且相信系统通过测试是很重要的。他还建议场景测试应该很容易进行评估。如果系统中存在问题，那么发布测试团队应该发现问题。

George 是心理健康保健的专业护士。他的职责之一是去病人家里进行访问，以确定治疗是否有效以及病人有没有产生药物副作用。

一天，George 进行病人家访，他登录到 Mentcare 系统中，使用该系统打印当天的家访日程表以及关于要访问的病人的摘要信息。他请求将这些病人的记录下载到自己的笔记本电脑上。系统提示他提供密钥短语从而在笔记本电脑上对记录进行加密。

George 要访问的一个病人是 Jim，他正在接受抗抑郁症药物治疗。Jim 感到药物正在帮助他改善状况但是认为药物有副作用——晚上会让他睡不着。George 检索 Jim 的记录，系统提示要求提供他的密钥短语来解密记录。他检查了所开的药物并且查询了药物的副作用。失眠是一个已知的副作用，因此他在 Jim 的记录中备注了这个问题，并且建议他去看门诊修改一下药方。Jim 同意了，George 输入了一条提示以提醒他回到诊所后跟医生做一下预约。George 结束了本次访问，系统重新对 Jim 的记录进行加密。

完成本次访问后，George 返回诊所，将所访问的病人记录上传到数据库中。系统为 George 生成一个联系清单，其中包括他要联系获取后续信息并进行诊所预约的病人。

图 8-10 Mentcare 系统的一个用户故事

作为一个来自 Mentcare 系统的一个可能的场景的例子，图 8-10 描述了系统用于家访时的一种使用方式。该场景测试了一些 Mentcare 系统的特征。

1. 通过登录系统进行身份认证；
2. 在笔记本电脑上下载和上传指定的病人记录；
3. 病人家访日程安排；
4. 在移动设备上对病人记录的加密和解密；
5. 病人记录检索和修改；
6. 连接保存了副作用信息的药物数据库；
7. 用于联系提示的系统。

如果你是一个发布测试人员，你要运行整个场景，扮演 George 的角色并且观察系统如何对不同的输入进行响应和表现。作为 George，你可能会故意犯一些错误，例如输入错误的密钥短语来解密记录。这可以检查系统对于错误的响应。你应该仔细地记录任何出现的问

题,包括性能问题。如果系统很慢,那么系统使用的方式会发生改变。例如,如果加密一个病人记录要花费很长时间,那么时间很紧的用户可能会跳过这个步骤。如果他们丢失了他们的笔记本,那么一个非授权的人就可以看到病人记录了。

在使用基于场景的方法时,通常都会在同一场景之中测试多个需求。因此,除了检查确认单个需求,也应检查确认这些需求的组合没有造成问题。

8.3.3 性能测试

一旦一个系统已经完全完成了集成,那么就可以对涌现性属性(例如性能和可靠性)进行测试了。必须设计性能测试以确保系统可以处理预计承受的负载。这通常都要运行一系列测试,其中会不断增加负载直到系统性能变得无法接受。

跟其他类型的测试一样,性能测试既关注展示系统满足其需求又关注发现系统中的问题和缺陷。为了测试性能需求是否得到满足,可能要构造一个运行说明。一个运行说明(见第11章)是一组反映将由系统进行处理的实际工作的测试。因此,如果一个系统中90%的事务属于类型A,5%属于类型B,而剩下的都是类型C、D、E,那么设计的运行说明中大部分测试要针对类型A。否则的话,就无法获得对于系统运行性能的准确测试。

当然,该方法并不是缺陷测试的最佳方法。经验表明发现缺陷的一种有效方式是围绕系统的限制设计测试。在性能测试中,这意味着以超出软件的设计极限的方式发出请求、给予系统压力。这被称为压力测试。

假设你正在测试一个事务处理系统,该系统被设计用于处理每秒不超过300笔交易。一开始你用每秒不到300笔事务的压力对系统进行测试。然后你再逐步增加系统的压力到超出每秒300笔事务,直到压力远超最大的设计系统负载并造成系统失效。

压力测试可以帮助你做以下两件事情。

1. 测试系统的失效行为。意料之外的一些事件组合可能会导致系统负载超出预计的最大负载的情形发生。在这些情况下,系统失效不应该造成数据丢失或非预期的用户服务损失。压力测试检查是确保系统在过载时导致系统“软性失效”而不是在过量负载下发生严重崩溃。

2. 揭示那些只会在系统满负载运转时出现的缺陷。虽然有人会说这些缺陷在正常使用时不太会导致系统失效,但是压力测试所模拟的非正常的情形组合仍然可能发生。

压力测试对于基于处理器网络的分布式系统尤其重要。这些系统经常会在负载很重时表现出严重的退化。网络会被不同处理器必须交换的协调数据所淹没。相关的处理过程会在等待所需要的来自于其他过程的数据时变得越来越慢。压力测试帮助发现何时系统会发生退化,这样就可以在系统中加入检查从而在超过临界点后拒绝新的事务请求。

8.4 用户测试

用户或客户测试是一个测试过程阶段,其中用户或测试提供他们对于系统测试的输入和建议。这可以是接受来自一个外部供应商的委托对一个系统进行正式的测试,也可以是一个非正式的过程,其中用户使用一个新的软件产品进行试验,看看是否喜欢这个产品,并检查确保该产品做了他们所需要的事情。用户测试很重要,即使已经进行了全面的系统测试和发布测试,来自用户工作环境的影响可能会对一个系统的可靠性、性能、可用性和鲁棒性产生重要的影响。

在实践中,对于系统开发者而言,复制系统的工作环境一般都是不可能的,因为开发者环境中的测试不可避免地会包含人为构造的成分。例如,一个计划在医院中使用的系统在一

个门诊环境中使用,其中还有其他一些事情在进行,例如,病人的危急状况以及与亲属的谈话等,这些都会影响系统的使用,但是开发者无法在他们的测试环境中包括这些东西。

有3种不同类型的用户测试。

1. α 测试,一组经挑选的软件用户与开发团队密切配合工作,对软件的早期发布进行测试;
2. β 测试,向一组更大规模的用户提供一个软件的发布版本,允许他们在上面进行试验,并将他们所发现的问题报告给系统开发者;
3. 验收测试,客户对一个系统进行测试以确定其是否已经就绪,是否可以从系统开发者那里接收过来并且在客户环境中进行部署。

在 α 测试中,用户和开发者在系统开发过程中一起工作来对系统进行测试。这意味着用户可以发现那些对开发测试团队来说不那么明显的问题。开发者实际上只能基于需求进行工作,但是这些需求经常没有反映其他影响软件实际使用的因素。因此,用户可以提供关于实践的信息,从而帮助设计出更加现实的测试。

α 测试经常在开发软件产品或应用程序时使用。对于这些产品有经验的用户可能会愿意参与 α 测试过程,因为这使得他们可以较早地了解他们可以进一步探索的关于新系统特性的信息,这也可以降低软件发生意料之外的变化从而干扰用户的业务的风险。 α 测试也可以在开发定制化软件的时候进行使用。敏捷开发方法追求用户参与开发过程,并且用户应当在设计系统测试时扮演关键的角色。

β 测试用于将一个软件系统的早期(有时候还未完成)发布提供给一个更大范围的客户和用户群体进行评估。 β 测试的参与人员可以是一组经过挑选的客户,他们一般都是系统的早期采用者。或者,也可以将软件公开提供出去,使任何有兴趣试用的人都可以使用。

β 测试主要用于要在许多不同的条件下使用的软件产品。 β 测试很重要,因为与定制化产品开发者不同,普通产品开发者无法限制软件的运行环境。产品开发者也无法知道并复制所有软件产品未来的使用条件。因此,可以利用 β 测试发现软件与它的运行环境特性之间的交互。 β 测试也可以成为某种形式的市场营销。客户通过 β 测试了解系统以及系统可以为他们做些什么。

验收测试是定制化系统开发的一个内在部分。客户使用他们自己的数据对一个系统进行测试,并决定是否可以从系统开发者那里接收系统。验收意味着应该为软件进行最终的付款。

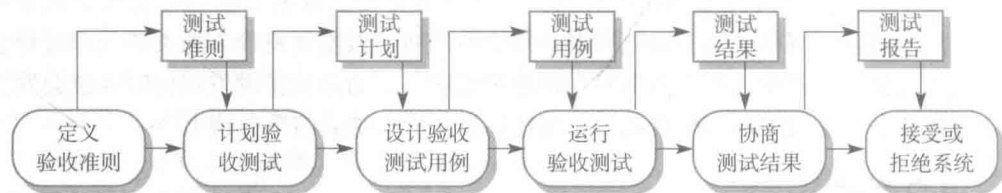


图 8-11 验收测试过程

图 8-11 描述了验收测试过程所包含的如下 6 个阶段。

1. 定义验收准则。理想情况下,这个阶段应当在开发过程中很早就发生,一般在系统的开发合同签署之前。验收准则应当成为系统开发合同的一部分,并且得到客户和开发方的批准。然而,在实践中,在开发过程中很早就定义准则是很困难的。此时可能还没有详细的需求,而且需求在开发过程中几乎总是会发生变化。

2. 计划验收测试。这个阶段会确定验收测试所需要的资源、时间和预算,并建立一个测

试日程表。验收测试计划还应当探讨所需要的需求覆盖度以及系统特征的测试顺序。计划中应当定义测试过程的风险,例如系统崩溃以及性能不足,并讨论如何缓解这些风险。

3. 设计验收测试用例。一旦建立验收测试准则后,就可以设计测试用例来检查系统是否可以接受。验收测试的目标应该是同时测试系统的功能和功能性特性。理想情况下这些测试应该提供完整的系统需求覆盖。然而在实践中,建立完整的客观验收准则是很困难的。因此,关于某个测试是否足以显示某个准则已经肯定被满足经常会存在争议。

4. 运行验收测试。协商达成一致的验收测试在系统上执行。理想情况下,这一步骤应当在系统将运行的真实环境中进行,然而这样可能会带来干扰并且不实际。因此,必须要建立一个用户测试环境来运行这些测试。这一过程很难自动化,因为验收测试中的一些部分会包含对最终用户和系统之间交互的测试。可能需要对最终用户进行培训。

5. 协商测试结果。所有已定义的验收测试都通过并且系统没有任何问题,这是不太可能的。如果真是这样的话,那么验收测试完成,系统就可以交接了。然而更常见的情况是会发现一些问题。这种情况下,开发者和客户必须进行协商以决定系统是否足够好并可以使用了。他们还必须就开发者将如何修复所发现的问题达成一致意见。

6. 接受或拒绝系统。这个阶段中开发者和客户要进行会谈以确定系统是否可以接受。如果系统不够好,还无法使用,那么需要进一步的开发来修复所发现的问题。一旦完成,还要重新进行验收测试阶段。

你可能会认为验收测试是一个清晰的合同问题。如果一个系统没有通过验收测试,那么系统不应该被接受,不应该进行付款。然而,现实往往更加复杂。客户由于立即部署可以获得一些好处,因此想尽快使用软件。他们可能会购买好了新的硬件,训练了工作人员,并且改变了他们的过程。他们可能会在软件还存在问题的情况下就愿意接受该软件,因为不使用软件的成本会比处理那些软件中的问题的成本更高。

因此,协商的结果可能是有条件地接受系统。客户会接受系统,这样部署可以开始。系统提供者同意修复紧急的问题并向客户尽快交付一个新版本。

在敏捷方法(例如极限编程)中,可能没有独立的验收测试活动。最终用户是开发团队的一部分(也就是说他是 α 测试人员),并且以用户故事的方式提供系统需求。他也会负责定义测试,这些测试决定了所开发的软件是否支持用户故事。因此,这些测试等同于验收测试。相关测试是自动化的,在用户故事的验收测试没有成功执行之前,开发不会向前推进。

当用户被纳入一个软件开发团队之中的时候,理想情况下他们应该是具有关于系统如何使用的一般知识的“典型”用户。然而,寻找这样的用户可能很难,因此验收测试实际上可能无法真正反映系统在实践中是如何使用的。而且,自动化测试的要求限制了测试交互式系统的灵活性。对于这样的系统,验收测试可能要求一些最终用户按照自己日常工作的方式来使用系统。因此,虽然原则上“用户参与”是一个很有吸引力的思想,但是它并不一定会得到高质量的系统测试。

用户参与敏捷团队会产生一些问题,因此许多公司将敏捷和更传统的测试方法混合使用。系统可能使用敏捷技术开发,但是完成一个主要的发布后会使用独立的验收测试来确定是否应该接受该系统。

要点

- 测试只能显示程序中存在错误,但这并不能说明程序中没有剩下的缺陷。

- 开发测试是软件开发团队的责任。一个独立的团队应该负责在一个系统发布给客户之前对其进行测试。在用户测试过程中，客户或系统用户提供测试数据并且确保测试是成功的。
- 开发测试包括：单元测试，会对各个对象和方法进行测试；构件测试，会测试相互关联的对象组；系统测试，会测试一部分或整个系统。
- 在测试软件时应该尽力把软件“弄坏”，这需要使用经验和指南来仔细选择特定类型的测试用例，这些类型的测试用例在其他系统中可以有效地发现缺陷。
- 只要有可能都应当尽量编写自动化测试。测试嵌入测试程序中，可以在每次对一个系统进行修改时运行这些程序。
- 测试先行的开发是一种开发方法，其中测试在要测试的代码开发之前就编写好了。然后可以对代码进行少量的修改并对代码进行重构，直到所有的测试都成功运行。
- 场景测试很有用，因为它复制了系统的实际使用方式。场景测试包括设计一个典型的使用场景，然后使用该场景得出测试用例。
- 验收测试是一个用户测试过程，其目的是确定软件是否足够好以至于可以进行部署并在原计划的运行环境中使用。

阅读推荐

《How to design practical test cases》是一篇关于如何设计测试用例的文章，作者来自于一家日本公司，该公司在交付高质量的软件方面享有良好的声誉。（T. Yamaura, IEEE Software, 15 (6), November 1998）<http://dx.doi.org/10.1109/52.730835>

《Test-driven development》是一期关于测试驱动开发的专刊，包括一篇对测试驱动开发的很好的宏观概览，以及一些关于测试驱动开发如何被用于不同类型的软件的经验文章。（IEEE Software, 24 (3) May/June 2007）

《Exploratory Software Testing》是一本关于软件测试的实践性而非理论性的书籍，对Whittaker在更早的一篇著作《How to Break Software》中的思想进行了进一步的拓展。作者提供了一组基于经验的软件测试指南。（J. A. Whittaker, 2009, Addison-Wesley）

《How Google Tests Software》是一本关于如何测试大规模基于云的系统的书，其中讨论了一整套与定制化软件应用相比的新挑战。虽然我并不认为Google的方法可以直接使用，但是本书中有一些关于大规模系统测试的有趣的教训。（J. Whittaker, J. Arbon, and J. Carollo, 2012, Addison-Wesley）

网站

本章的PPT：<http://software-engineering-book.com/slides/chap8/>

支持视频的链接：<http://software-engineering-book.com/videos/implementation-and-evolution/>

练习

- 8.1 为什么对于一个程序而言没必要在交付给客户之前保证完全没有缺陷？
- 8.2 为什么测试只能表明错误的存在，而不是显示没有错误存在？
- 8.3 一些人认为开发人员不应该参与测试自己的代码，所有的测试责任都应该由一个独立的团队来承担。给出关于开发人员自己测试的支持和反对两方面的论点。

- 8.4 你被安排测试“Paragraph”对象中一个名为 catWhiteSpace 的方法，该方法在一个段落里面将所有连续的空格符替换成单个的空格符。为这个例子确定测试划分，并且为 catWhiteSpace 方法设计一组测试。
- 8.5 什么是回归测试？解释该如何使用自动化测试以及 JUnit 这样的测试框架来简化回归测试。
- 8.6 Mentcare 系统是通过成品信息系统进行适配性修改得到的。测试这样一个系统与测试使用 Java 这样的面向对象语言开发的软件有什么区别？
- 8.7 编写一个可以用于为野外气象站系统设计测试的场景。
- 8.8 你是如何理解术语“压力测试”的？谈一谈如何对 Mentcare 系统进行压力测试。
- 8.9 在测试过程的早期阶段让用户参加发布测试有什么好处？这种用户参与有什么不好的地方吗？
- 8.10 系统测试的一个常用方法是测试系统直到测试预算耗尽，然后将系统交付给客户。针对交付给外部客户的系统，讨论这种方法的道德问题。

参考文献

- Andrea, J. 2007. “Envisioning the Next Generation of Functional Testing Tools.” *IEEE Software* 24 (3): 58–65. doi:10.1109/MS.2007.73.
- Beck, K. 2002. *Test Driven Development: By Example*. Boston: Addison-Wesley.
- Bezier, B. 1990. *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold.
- Boehm, B. W. 1979. “Software Engineering; R & D Trends and Defense Needs.” In *Research Directions in Software Technology*, edited by P. Wegner, 1–9. Cambridge, MA: MIT Press.
- Cusamano, M., and R. W. Selby. 1998. *Microsoft Secrets*. New York: Simon & Schuster.
- Dijkstra, E. W. 1972. “The Humble Programmer.” *Comm. ACM* 15 (10): 859–866. doi:10.1145/355604.361591.
- Fagan, M. E. 1976. “Design and Code Inspections to Reduce Errors in Program Development.” *IBM Systems J.* 15 (3): 182–211.
- Jeffries, R., and G. Melnik. 2007. “TDD: The Art of Fearless Programming.” *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.
- Kaner, C. 2003. “An Introduction to Scenario Testing.” *Software Testing and Quality Engineering* (October 2003).
- Lutz, R. R. 1993. “Analysing Software Requirements Errors in Safety-Critical Embedded Systems.” In *RE’93*, 126–133. San Diego CA: IEEE. doi:10.1109/ISRE.1993.324825.
- Martin, R. C. 2007. “Professionalism and Test-Driven Development.” *IEEE Software* 24 (3): 32–36. doi:10.1109/MS.2007.85.
- Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.
- Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action*, 2nd ed. Greenwich, CT: Manning Publications.
- Whittaker, J. A. 2009. *Exploratory Software Testing*. Boston: Addison-Wesley.

软件演化

目标

本章的目标是解释为什么软件演化是软件工程中一个重要组成部分，并且介绍维护大型软件系统所面临的挑战。阅读完本章后，你将：

- 理解软件系统如果要保持有用就必须不断适应和演化，软件变更和演化应当成为软件工程的一个有机部分；
- 理解遗留系统的含义以及这些系统为什么对业务很重要；
- 了解如何对遗留系统进行评估，从而决定是应该抛弃、维护、再工程还是更换这些系统；
- 学习不同类型的软件维护以及影响维护费用的因素。

大型软件系统通常有一个很长的生命周期。例如，大型的军事或者基础设施系统（例如航空交通管制系统）可能拥有 30 年或者更长的生命周期。业务系统的生命周期通常也会在 10 年以上。企业软件的成本很高，所以一个公司会使用一个软件系统很多年来收回前期对软件产品的投资。因此，那些很多年以前就取得成功的软件和产品，依然会定期发布新版本。例如，微软 Word 的第一个版本早在 1983 年就问世了，并且在此后的 30 年间不断进行着更新。

在系统开发完成后，如果要使其继续有用，对它进行修改是不可避免的。由于业务变更和用户期待的改变，使得对已有系统的新需求浮现出来。由于各种原因，软件的某些部分需要修改，例如，修复运行中发现的错误、适应新的运行平台、提升性能或其他的非功能特性。所有这些都意味着在系统交付后，软件系统总是在不断演化以满足变更的需求。

企业必须不断改进他们的软件以确保他们能够持续从中获得价值。他们的系统已经成为组织的重要经营资产，必须投资进行系统变更以保持其价值。通常，大多数大型公司在维护系统上的开支要比在系统开发上的开支多很多。历史统计数据（Lientz and Swanson 1980; Erlikh 2000）表明，60% ~ 90% 的软件花费是演化花费。Jones（Jones 2006）的调查结果也显示在 2006 年美国的 IT 企业中，有大约 75% 的工作都与软件演化有关，并且在可预见的未来这一数字没有下降的倾向。

对于一些大规模的企业系统来说，软件演化的代价极其昂贵，因为一个相对独立的系统往往是“由系统构建的（整体）系统”中的一部分。在这种情况下，进行变更需要考虑的不仅仅是系统本身的影响，还有对全局的（整体）系统中其他部分的影响。因此，对某个系统进行变更通常意味着还需要对其他系统也进行相应的修改，以适应环境的变更。

因此，在确认和分析变更对系统本身产生影响的同时，还需要评估它会对运行环境中的其他系统产生怎样的影响。Hopkins 和 Jenkins（Hopkins and Jenkins 2008）发明了一个术语棕地软件开发（brownfield software development）来描述这种情况，即软件系统在其开发和管理所处的环境中依赖于其他许多软件系统。

已安装的系统，随着业务和它的环境的改变，其需求也随之改变。因此，系统通常会定期发布新版本以实现改变和更新。因此，软件工程是一个贯穿系统生命周期的，由需求、设计、实现、测试组成的螺旋过程（见图 9-1）：开始于系统的第 1 个发布版本的创建；一旦交付使用，变更提出，则第 2 个发布版本的开发立刻开始。事实上，甚至于在系统部署之前，演化的需求可能已经变得很明显，以至于在目前的版本发布之前，软件的后继版本就已在开发中了。

最近 10 年，软件螺旋式迭代的周期正变得越来越短。在互联网普及之前，软件新版本发布的周期大约是 2 ~ 3 年。而如今面对激烈的竞争和及时的用户反馈，很多软件与 Web 应用的更新周期甚至可以短至数周。

这个软件演化模型是针对一个专门组织既负责最初的软件开发又负责以后的软件演化这种情况的。当软件开发完成后，团队需要将工作无缝衔接到软件演化上，并且开发过程中使用的过程和方法会始终贯穿于软件的整个生命周期当中。大部分打包的软件产品是按这种方式开发的。

对于定制化软件，通常使用另一种不同的方法。一个软件公司为客户开发软件，然后由客户自己的开发团队接管这个系统，即由他们负责软件演化。还有另一种选择是，软件用户和另外一家公司签订一个单独的合同，要求其负责系统的支持和演化。

在这种情况下，螺旋进程经常是不连续的。需求和设计文档不能从一个公司传递到另一家公司。公司可能通过合并或重组继承来自其他公司的软件，于是发现需求和设计文档都必须进行变更。当从开发到演化的转换不是无缝衔接时，软件移交之后的变更过程被称为“软件维护”。正如在本章的后面将会谈到的，维护包括额外的过程活动，例如，除正常的软件开发活动外的程序理解。

Rajlich 和 Bennett (Rajlich and Bennett 2000) 提出一个软件演化生命周期的另一种视图，如图 9-2 所示。在这个模型中，他们把演化和维修区别开来。演化阶段涉及软件体系结构和功能性的重大改变；而维修阶段所做的都是一些相对较小但必不可少的修改。

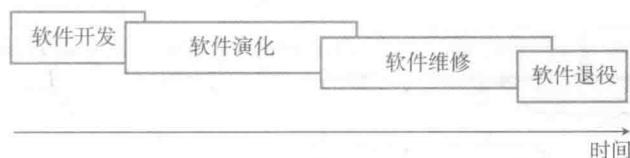


图 9-2 演化和维修

根据 Rajlich 和 Bennett 提出的模型，当软件初步被成功使用时，许多关于需求变化的提议会被采纳和实现。这是演化阶段。但是，随着软件被更改，它的结构也在退化，更改的成本也变得越来越。这种情况通常发生在使用数年之后，其间也有其他环境的改变，例如硬件或者操作系统。在生命周期的某些阶段软件会到达一个转折点，在此之后软件进行重大

的改变以及实现新需求都会变得越来越不划算。这时，软件从演化阶段转为维修阶段。

在维修阶段，软件仍然是可用的并且一直在使用着，只有一些小的局部的调整需要做。在这个阶段，公司经常会考虑怎样将软件更换掉。在最后阶段，软件仍然在使用，但是只有少量必需的变更会被执行，因而用户需要想办法去绕过发现的某些问题。最终，软件退役并完全退出使用。这通常会包含将旧系统的数据迁移到新的替代系统上，其中会包含一些额外的成本。

9.1 演化过程

和其他很多软件过程一样，软件变更与软件演化并不存在绝对的标准。软件演化过程在相当程度上依赖于所维护的软件的不同类型和参与开发过程的组织和人。对于有些类型的系统，例如移动应用，演化可能是一种非正式的过程，变更请求大部分来自于系统用户和开发者的交流。而对于其他类型的系统，例如嵌入式关键性系统，这却是一个在每个阶段都产生结构化文档的正式过程。

在所有的组织中，正式或非正式的系统变更建议都是系统演化的动力。在变更建议中，个人或团队会对现有系统的改进或更新提出自己的建议。这些变更建议可能包括在发布的版本中还没有实现的已有需求、新的需求请求、系统所有者的补丁要求和来自系统开发团队的改进软件的新想法和建议。变更识别的过程和系统演化是循环和持续的，并且贯穿于系统的生命周期（见图 9-3）。

在接受变更建议之前，需要对软件进行分析，以确定哪些构件需要更改。该分析允许评估变更的成本和影响。这是变更管理的一般过程的一部分，它还应确保每个系统版本中都包含正确版本的构件。第 25 章将讨论变更和配置管理。



图 9-3 变更识别和演化过程

图 9-4 摘自 Arthur (1988)，展示了演化过程的概况。演化过程包括变更分析的基础活动、版本规划、系统实现和对客户发布。通过评估这些变更的花费与影响，来发现系统在多大程度上受到影响以及实现变更可能花费多少。

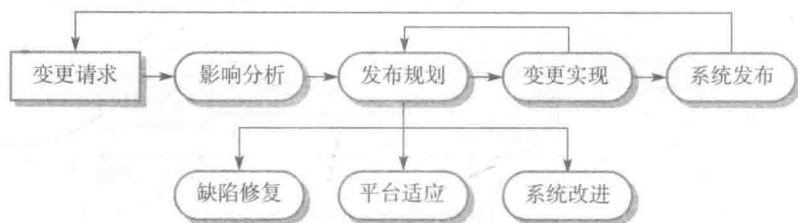


图 9-4 软件演化过程模型的通用模型

如果所提议的变更被接受，则会规划一个新的系统发布版本。在发布规划期间，考虑所有提出的变更建议（故障修复、适应和新功能），然后决定在系统的下一个版本中实现哪些变更。接下来实现并确认变更，并发布一个新的系统版本。这之后，该过程会进入下一个迭

代，考虑针对下一个发布所提出的一组新的变更。

在有些情况下开发和演化被集成到了一起，此时变更实现只是开发过程的一个迭代。针对系统的各种修改进行设计、实现和测试。初始开发和演化之间唯一的区别是，交付后的客户反馈在规划一个应用的新发布版本时必须进行考虑。

而当开发和变更过程是由不同的团队各自负责时，最重要的不同点在于，负责变更工作的团队首先要理解源代码的实现方式。在这个阶段，要了解程序是怎样构造和怎样实现它的功能的。在实现一个变更时，我们要利用以上的理解来确保实现的变更不会对现有系统产生不利影响。

如果需求规格说明和设计文档是存在的，那么在实现变更后，应该对其进行修改以反映系统的变化情况（见图 9-5）。新的软件需求也应该被记录下来，用于分析和确认。如果设计文档中还包含 UML 模型，那么也应该对其进行相应的修改。这些工作或许应该在需求变更的分析阶段进行，这样就能有效地评估变更的花费和影响。

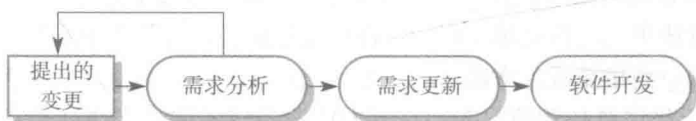


图 9-5 变更实现

变更请求有时关系到需要立刻解决的一些系统问题。以下这些原因可能会导致出现紧急的变更。

1. 一个严重的系统缺陷被发现，必须修复该缺陷以使常规的系统运行可以继续或者解决一个严重的信息安全漏洞。
2. 如果系统操作环境的变更中有不期望的情况发生，那么就会破坏正常的操作。
3. 如果系统上运行的业务有未预料到的改变发生，例如，有新的竞争对手出现或者有新的法律生效并影响到了系统。

在这些情况下，就需要做出快速的变更，也意味着不能遵循正常的变更分析过程——不是按正常的过程去修改需求和设计，而是要对程序进行紧急的修补来解决突发问题（见图 9-6）。这样做的危险是使需求、软件设计和代码逐渐变得不一致。尽管你打算记录需求和设计方面的变更，却可能又有一个急需的修补在等你完成。它的优先权高于文档记录。最后，最初的变更被遗忘了，系统文档与代码再也不能一致了。这其中存在的一个问题就是，维护大量的文档与敏捷方法中尽可能减少文档数量的基本目标相冲突。



图 9-6 紧急修复过程

紧急系统修补通常需要尽可能快地完成。应当选择一种快速的可行方案，而不是选择一个能保证系统结构最好的方案。这就加速了软件的老化过程，并使未来的变更计划更困难，维护费用上升。在理想的情况下，在紧急修补完成后，最好应该再对代码进行重构，以提高代码质量和避免软件老化。当然，修补代码可能再次得到利用。如果我们有更多的分析时间的话，就有可能找到此问题的另一个更好的解决方案。

敏捷方法和过程（在第3章中讨论过）也许会在程序演化和程序开发中用到。实际上，因为这些方法是基于增量开发的，敏捷开发向交付后演化的转变应当是无缝衔接的。

然而，当一个开发团队向另外一个负责演化的团队交接时，有以下两种问题则可能出现。

1. 开发团队运用了敏捷方法，但是演化团队却不熟悉敏捷方法而选择了一个计划驱动的方法。演化团队可能期望详细的文档来支持演化工作的进行，而这恰恰是敏捷方法所不能提供的。可能没有关于系统的一个明确的描述以供变更时使用。

2. 计划驱动的方法被用于开发过程，而演化团队选择使用敏捷方法。这种情况下，演化团队可能不得不从头开发自动化的测试，而且不会有像敏捷开发过程中所期待的系统代码重构和简化。这样，在采用敏捷开发过程之前，可能要求使用一些再工程方法来提升其代码质量。

诸如测试驱动开发和自动回归测试这样的敏捷方法可以在系统变更时使用。很多系统变更会以用户故事的方式被提出，这些用户的参与可以为确定系统变更的优先级顺序提供极大的帮助。Scrum方法中关注待处理工作的特性也能帮助确定最重要的系统变更。总之，软件演化过程仍然需要一些敏捷开发方法。

然而，当用于程序维护和演化时，在开发中所使用的敏捷方法必须进行一些修改。在实践中让用户参与开发团队是不太可能的，因为变更提议来自范围很广的利益相关者。短开发周期可能不得不被打断来处理紧急修复，发布之间的差距可能不得不拉大以避免干扰运行过程。

9.2 遗留系统

大型企业一般在20世纪60年代开始将他们的运营计算机化，因此在过去的大约50年中，越来越多的软件系统被引入到这些企业中。这些系统中许多都已经随着业务的变更和演化被替换掉了（有时候会多次替换）。然而，很多旧系统仍然还在使用并且在业务的运行中扮演着重要的角色。这些旧软件系统有时候被称为遗留系统。

遗留系统是比较老的系统，它们依赖于一些在新系统开发中不再使用的语言和技术。典型情况下，它们已经被维护了很长一段时间，它们的结构可能已经由于所做的修改而发生了退化。遗留软件可能依赖于更老的硬件，例如主机计算机，而且可能与遗留过程和规程相关联。这些软件可能无法通过变更来适应更加有效的业务过程，因为遗留软件无法被修改来支持新的过程。

遗留系统不仅仅是软件系统，还是更广阔的社会技术系统，其中包括硬件、软件、各种库，以及其他支持性的软件和业务过程。图9-7描述了一个遗留系统的逻辑成分以及它们的关系。

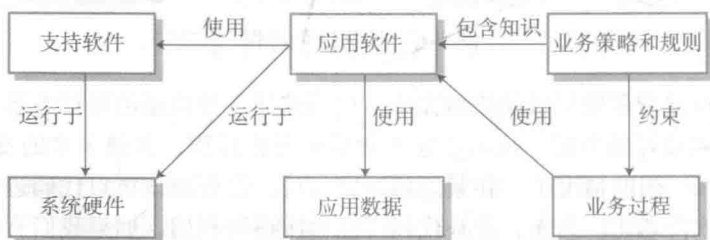


图 9-7 遗留系统中的元素

1. 系统硬件。遗留系统当初开发的时候可能是针对一些老的硬件编写的, 这些硬件现在已经不再可用、维护很昂贵, 并且可能与当前的组织信息技术采购政策不相符。

2. 支持软件。遗留系统可能依赖于一系列支持软件, 从操作系统以及硬件厂商提供的一些工具到用于系统开发的编译器。这些软件也可能过时了, 它们原来的提供者可能不再提供支持了。

3. 应用软件。提供业务服务的应用系统通常由一些应用程序组成, 这些应用程序是在不同的时间开发的。其中一些程序将会同时成为其他应用软件系统的一部分。

4. 应用数据。这些数据是应用系统处理的。许多遗留系统都在漫长的系统生命周期中积累了大量的数据。这些数据可能会不一致, 可能在多个文件中重复出现, 还可能分散在一些不同的数据库中。

5. 业务过程。这些过程在业务中使用以实现一些业务目标。一个业务过程的例子包括一个保险公司发布一个保险政策; 一家制造业企业的业务过程会包括如何接受产品订单并且安排相关的制造过程。业务过程可能会围绕一个遗留系统设计并且受该系统提供的功能的制约。

6. 业务政策和规则。这些政策和规则包括关于业务应当如何开展的定义以及对于业务的约束。遗留应用系统的使用可能会蕴含在这些政策和规则中。

另一种看待这些遗留系统组成成分的方式是分一系列的层, 如图 9-8 所示。

每一层都依赖于紧挨着的下一层并且与该层有接口。如果能够保持接口不变, 那么应该可以在一层之内进行修改而不会影响到相邻的层。然而, 在实践中这种封装过于简单化, 对于系统中某一层的修改可能会要求对所改变的那一层的上一层和下一层都进行修改。其中的原因包括以下 3 个方面。

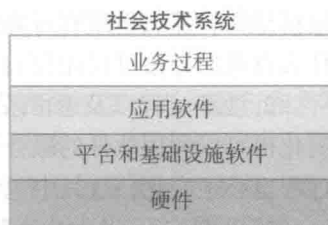


图 9-8 遗留系统分层

1. 修改系统中的某一层可能会引入新的工具, 系统中更高的层可能会进行修改以利用这些新的工具。例如, 在软件层引入一个新的数据库可能会包括通过 Web 浏览器访问数据的工具, 因此业务过程可能会进行修改以利用这一工具。

2. 修改软件可能会使系统变慢, 从而需要新的硬件来改进系统性能。来自新硬件的性能提升接下来又意味着原来不现实的对软件进一步的修改成为可能。

3. 维护硬件接口经常是不可能的, 特别是当新的硬件引入后, 这在嵌入式系统中尤其棘手, 在这类系统中软件和硬件之间存在着紧密的耦合。为了有效利用新的硬件, 应用软件需要进行大幅度的修改。

想确切知道有多少遗留代码仍然在使用是很难的, 但是作为一个大致的指示器, 工业界估计当前的业务系统中存在超过 2 000 亿行 COBOL 代码。COBOL 是一种被设计用于编写业务系统的编程语言, 在 20 世纪 60 年代到 20 世纪 90 年代之间都是主要的业务开发语言, 特别是在财务领域 (Mitchell 2012)。这些程序仍然在有效和高效地工作, 使用这些程序的企业觉得没有必要修改它们。然而, 他们面临的一个主要问题是随着原来的系统开发者退休, 越来越难找到 COBOL 程序员。大学不再教授 COBOL, 更年轻的软件工程师对使用现代的语言编程更感兴趣。

技能的缺乏仅仅是维护业务遗留系统的众多问题中的一个。其他问题还包括: 信息安全漏洞, 因为这些系统是在互联网广泛使用之前开发的; 与用现代编程语言编写的系统进行接

口交互时所面临的问题。最初的软件工具供应商可能已经退出相关业务，或者不再维护当初用于开发系统的支持工具。系统硬件可能会过时并且维护越来越贵。

那么为什么企业没有简单地用更加现代化的同类系统替换这些系统呢？对于这个问题的一个简单的回答是这样做太贵并且风险太大。如果一个遗留系统还能够有效工作，那么更换系统的成本可能会超过来自新系统支持成本降低所节省的费用。将遗留系统废弃掉并用更加现代的软件来替换它们，很可能会使事情走上歧途，或者出现新系统无法满足业务需要的情况。管理层试图将这种风险最小化，因此不想面对新软件系统所带来的不确定性。

当我参加一个大型组织的遗留系统替换项目时，我发现了遗留系统替换的一些问题。这家企业使用了超过 150 个遗留系统来支撑其业务。该企业决定将所有这些系统替换成一个集中维护的 ERP 系统。由于一系列业务和技术原因，新系统开发失败了，它没有提供所承诺的改进。在花费了 1000 万英镑后，新系统中只有一小部分可以运行，其运转比所替换的老系统更加低效。用户继续使用老系统，但是却无法将这些与所实现的新系统的部分集成起来，因此还需要额外的人工处理。

以下是关于用新系统替换遗留系统为什么那么昂贵以及风险高的几点原因。

1. 遗留系统很少有一个完备的规格说明。最初的规格说明可能已经遗失。如果存在一个规格说明，那么也很有可能并没有及时更新以反映在该系统上实施的所有修改。因此，没有什么直观的办法可以让我们去刻画一个与正在使用的系统功能上等价的新系统。

2. 业务过程以及遗留系统的运行方式经常不可避免地交织在一起。这些过程很可能已经演化得可以利用软件的服务，同时对软件的缺点进行了变通。如果系统被替换掉，那么这些过程也必须变化，这其中包含不可预测的成本以及后果。

3. 重要的业务规则可能会蕴含在软件中，并且可能并没有进行专门的文档描述。一条业务规则是一个适用于一些业务功能的约束，违反约束可能会对业务造成无法预计的后果。例如，一个保险公司可能会将他们评估一个政策应用风险的规则蕴含在其软件中。如果这些规则没有得到保持，那么这家公司可能会接受未来导致昂贵索赔的高风险政策。

4. 新的软件开发从内来讲也是充满风险的，因此新系统中可能会存在无法预见的问题。新系统可能会无法按时以及按照预计的价格交付。

继续保持使用遗留系统可以避免系统替换的风险，但是随着系统逐渐老化，在已有的软件中进行修改可能会变得越来越昂贵。对使用多年的遗留软件系统的修改尤其困难，这主要是由于以下这些原因。

1. 程序风格和使用习惯不一致，因为系统的修改是由不同的人负责的，这加剧了理解系统代码的困难。

2. 系统的一部分或全部可能是用过时的编程语言实现的。找到懂这些语言的人可能会很难。因此，可能需要花费昂贵的代价寻找系统维护外包。

3. 系统文档化经常不充分而且过时。有时候，系统唯一可用的文档就是系统源代码。

4. 多年的维护通常会使得系统结构退化，使理解系统越来越难。新的程序可能是通过临时的方式添加进去并与其他部分交互的。

5. 系统可能针对空间利用或执行速度进行了优化，从而使系统在更老且更慢的硬件上能够有效运行。这通常都会使用特定的机器和语言进行优化，这些优化通常都会导致软件难以理解。这对于学习现代软件工程技术且不懂在程序中使用的那些编程技巧的程序员而言是个

问题。

6. 系统所处理的数据可能会保存在多个结构不匹配的文件里。其中可能存在数据重复, 数据本身可能会过时、不准确、不完整。可能会使用多个来自不同供应商的数据库。

到了某一阶段, 管理和维护遗留系统的成本变得如此之高, 以至于不得不将现有系统替换为一个新系统。在下一节中, 将讨论一个做出这类替换决策的系统性的决策方法。

9.2.1 遗留系统管理

对于使用现代软件工程过程(例如, 敏捷开发和软件产品线)来开发的新软件系统而言, 规划如何将系统开发和演化集成到一起是可能的。越来越多的公司开始认识到系统开发过程是一个全生命周期的过程, 人为地将软件开发和软件维护分开并不好。但是, 仍然存在许多遗留系统, 它们是十分重要的业务系统。我们必须扩展和调整它们以适应变化中的电子商务环境。

大多数组织拥有很多遗留系统, 对这些遗留系统的维护和更新的资金又非常有限, 这时候就需要对投资做精心的安排, 以期获得最佳的回报。这就要求组织先对遗留系统进行现实的评估, 然后做出适当的决策, 有以下4种基本选择。

1. 彻底废弃这个系统。当系统不能对业务过程产生有效的作用时, 应该选择这个方案。这种情况一般发生在系统安装之后, 业务过程已经改变, 新的业务过程不再依赖于遗留系统。

2. 不再大幅修改系统仅保持常规维护。当一个系统仍然有存在的必要, 系统运行相当平稳, 而且用户没有提出太多对系统变更的要求时, 应该选择这个方案。

3. 对系统进行再工程以改善其可维护性。当系统质量由于经常性的变更已经下降, 而且仍然需要做经常性的变更时, 应该选择这个方案。这个过程应当包括开发新的构件接口, 从而使最初的系统能和其他的新系统协同工作。

4. 用一个新的系统代替整个或部分系统。当其他因素(例如, 新的硬件已经使旧系统无法继续运行, 或者有现成的产品可以使用)使新系统的开发成本非常合理时, 就应做出此种选择。在很多情况下, 可以采用演化替换策略, 用现有的系统替换主要的系统构件, 并且在可能的情况下继续使用其他构件。

在评估一个遗留系统的时候, 必须同时从业务和技术两个视角来看待它(Warren 1998)。从业务的视角看, 必须对该系统的业务价值做出评估。从技术的视角看, 必须对应用软件、系统支持软件和硬件质量做出评估。根据这两个方面得到的评估结果就能决定应该对遗留系统采取何种策略了。

例如, 假设某组织有10个遗留系统。首先评估每个系统的质量和业务价值, 然后画在一张图中, 比较系统在这两方面的相对水平, 如图9-9所示。从图中可以看出有以下4类系统。

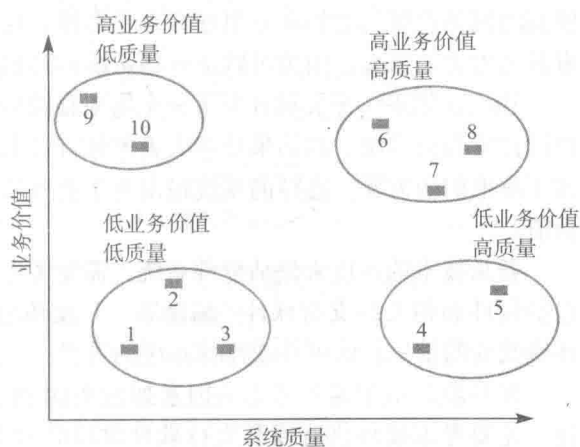


图9-9 一个遗留系统评估的例子

1. 低质量、低业务价值系统。保持这些系统继续运转的费用很高、回报率很低。这类系统是抛弃的候选对象。

2. 低质量、高业务价值系统。这些系统正在为业务做出重要贡献，不能抛弃。不过，其低质量意味着运行的成本很高，所以这类系统有待于转换或者以合适的系统替代。

3. 高质量、低业务价值系统。这些系统对业务的贡献很小，但是维护费用较低。不值得冒险去替换这种系统，可以继续进行的系统维护，也可以抛弃。

4. 高质量、高业务价值系统。这种系统必须保持运转，其高质量说明无须对其投资进行转换或更换。正常的系统维护应该继续进行。

为了评估一个系统的业务价值，我们必须假设是系统的所有者，比如最终用户和他们的经理，对系统提出以下4个方面的问题。

1. 系统的使用。如果系统只是偶尔被使用或被很少一部分人使用，那么它们含有较低的业务价值。遗留系统可能是为满足某些业务需要而开发的，但现在业务改变了，或者有其他更有效的方法满足要求。然而，一定要注意不经常使用但却重要的系统。例如，在大学里，一个学生的注册系统可能仅是在每学年的开始时才使用。但是，它却是具有高业务价值的必不可少的系统。

2. 支持的业务流程。当引入一个系统时，我们就要设计业务流程来开发系统的能力。如果遗留系统难以改变，那么改变这些业务流程就是不可能的。但是，当环境变化时，原来的业务流程可能会变得不可用。因而，由于不能引入新的流程，很可能使一个系统的业务价值降低。

3. 系统的可靠性。系统可靠性不仅仅是一个技术问题，也是一个业务问题。如果系统不可靠，而且问题直接影响商业用户，或者让业务处理中的人从别的任务转过来解决这些问题，那么系统的业务价值较低。

4. 系统的输出。这里的关键问题是系统输出对于成功的业务功能的重要性。如果业务依赖于这些输出，那么系统就含有高的业务价值。相反，如果这些输出可以用某种方法很容易地产生，或者系统产生的输出很少被使用，那么它的业务价值就很低。

例如，有一家公司提供一个旅行预定系统，负责安排旅行的人员可以通过此系统向认可的旅行代理人发出订单，该公司然后得到预订凭证和发票。但是，业务价值评估可能显示在总的旅行订单中只有相当小的一部分使用了此系统。负责旅行安排的人们发现，通过旅行提供商的网站直接与它们联系更便宜也更方便。这个系统还将继续使用，但是保留这个系统没有什么太大的意义，因为可以从外部系统得到相同的功能。

相反，如果一个公司开发了一个用来跟踪所有客户以前的订单记录并能自动生成提醒客户续订货物的系统。其结果是有大量的续订订单，而且使客户倍感满意，因为他们感到供应商了解他们的需要。这样的系统输出对于业务来说是非常重要的，因此该系统有很高的业务价值。

若从技术的角度来评估软件系统，需要考虑应用系统自身和系统的运行环境。运行环境包括硬件和相关的支持软件（编译器、开发环境等）。环境很重要，是因为很多系统变更是环境改变的结果，如硬件或操作系统的升级。

在环境评估中需要考虑的因素如图9-10所示。注意这些因素并不是环境的所有技术特性，还要考虑硬件供应商和支持软件供应商的可依赖性。如果这些供应商不再从事此项业务，他们就不会再提供对系统维护的支持。

因 素	问 题
供应商稳定性	供应商还存在吗？供应商财务上是否稳定？供应商是否会继续存在？如果供应商不再从事相关业务，是否有其他人可以维护该系统
失效率	所报告的硬件失效率是否很高？支持软件是否崩溃并迫使系统重启
年限	硬件和软件已使用多少年？硬件和支持软件越老就越落后。它们可能仍然正确运行，但转向一个更现代的系统会带来显著的经济和业务上的回报
性能	系统的性能是否足够？性能问题对系统用户是否有显著的影响
支持需求	硬件和软件需要什么样的本地支持？如果这些支持成本很高，那么可能值得考虑下系统替换
维护成本	硬件维护和支持软件许可证的成本是什么？比较老的硬件的维护成本可能会比现代化的系统更高。支持软件可能会有较高的年度许可证成本
互操作性	该系统与其他系统的接口交互是否存在问题？例如，编译器是否可以与当前的操作系统版本一起使用

图 9-10 环境评估中所使用的因素

如果可能的话，在对环境评估的过程中，应该给出系统的度量和系统的维护过程。有用的数据的例子包括：维护系统硬件和支持软件的花费，在某个确定时间内硬件缺陷发生的次数，对系统支持软件打补丁和修改发生的频度等。

为了评估应用系统的技术质量，我们要评估一系列因素（见图 9-11），这些因素主要关于系统的可靠性、维护系统的困难以及系统的文档建立。我们还要收集量化的系统数据来帮助我们对系统质量做出判断。质量评估中可能用到的数据有：

- 1. 请求系统变更的数量。系统变更更容易造成系统结构的损坏，并为进一步变更增加了难度。这个数值越高，系统的质量也越低。
- 2. 用户界面数量。这对于基于表格的系统来说是一个重要的因素。在这种系统中，每一张表格都可以看作一个独立的用户界面。界面的数量越多，越容易发生界面的不一致和冗余。
- 3. 系统使用的数据量。使用的数据量（文件的数量、数据库的规模等）越大，就越有可能出现降低系统质量的数据不一致性。当数据经过长时间的积累，不可避免地会存在错误和不一致。清洗旧数据是一个非常昂贵和耗时的过程。

因 素	问 题
可理解性	理解当前系统的源代码有多难？所使用的控制结构有多复杂？变量的名字是否能反映它们的功能？
文档	可用的系统文档有哪些？文档是否齐全、一致并进行了及时更新？
数据	系统是否存在一个显式的数据模型？文件之间的数据在多大程度上存在重复？系统所使用的数据是否及时更新而且一致？
性能	应用的性能是否足够？性能问题对系统用户是否有显著的影响？
编程语言	开发该系统所使用的编程语言是否存在现代的编译器？该编程语言是否仍然被用于新系统开发？
配置管理	系统的所有部分的所有版本是否都由一个配置管理系统进行管理？当前系统所使用的构件的版本是否存在一个明确的描述？
测试数据	系统的测试数据是否存在？当新特性被增加到系统中时，所执行的回归测试是否有记录？
人员技能	具有维护该应用的技能的人是否存在？对于该系统有经验的人是否存在？

图 9-11 应用评估中所使用的因素

理论上说,应该根据客观的评估结果做出处理遗留系统的决定。然而在许多情况下,做出的决定并不是客观的,而是基于组织或政治上的考虑。举例来说,如果两个组织合并,其中一个组织在政治上地位显赫,它的系统就会被保留下来,另一个组织的系统显然被丢弃。如果高层管理者做出向新硬件平台迁移的决定,那么应用就需要被替换。如果在某一特定的年度内没有用于系统转换的预算,那么系统维护还要继续下去,尽管这将带来较高的长期成本。

9.3 软件维护

当软件交付后,软件维护就成为软件变更的一个常规过程。变更可以是一种更正代码错误的简单变更,也可以是更正设计错误的较大范围的变更,还可以是对描述错误进行修正或提供新需求这样的重大改进。变更的实现是修改已有的系统构件以及在必要的地方添加新构件到系统中。

有3种不同类型的软件维护。

1. 修复软件缺陷。通常改正代码错误费用相对较低,改正设计错误费用就高得多,因为要重写很多程序构件。需求错误的更正费用更高,因为必须对系统进行大量的重设计。

2. 使软件适应不同操作环境。在系统环境的某些方面发生改变时,需要进行这种类型的维护。环境上的改变包括硬件变化、操作系统平台的变化或其他的支持软件的变化。为了适应这些环境变化必须修改应用系统。

3. 增加或修改系统功能。当系统需求随着组织因素或业务改变而变更的时候,这种类型的维护就是必要的了。这时系统需要变更的范围通常要比其他类型的维护要大得多。



程序演化动力学

程序演化动力学是针对演化中的软件系统的研究,由Manny Lehman和Les Belady两位先驱在20世纪70年代开展的。这一研究得出了所谓的Lehman定律,被认为适用于所有的大规模软件系统。这些定律中最重要的一些内容包括:

1. 一个程序如果要保持有用就必须持续变更;
2. 随着程序的不断变更,它的结构在退化;
3. 在一个程序的生命周期中,变化率大致不变,并且独立于可用的资源;
4. 在一个系统的每一个发布中的增量修改大致不变;
5. 新的功能必须加入系统中以增加用户的满意度。

<http://software-engineering-book.com/web/program-evolution-dynamics/>

在实际过程中,这些不同类型的维护之间没有一个明确的界限。在使软件适应一个新环境的时候,可能需要增加新的功能来充分利用环境提供的服务。软件缺陷可能是因为系统以一种未预料的方式下使用才得以暴露,修正这类缺陷的最好方法是改变系统以适应它们的工作方式。

尽管维护的不同类型得到了普遍认可,但有时可能会使用其他不同的分类名称。人们广泛使用的“纠正性维护”是指缺陷修补维护;“适应性维护”有时是指为适应新环境所做的维

护,而有时又是指对新需求的适应性维护;“完善性维护”有时是指实现一个新需求来完善软件,而有时又是指保留系统的功能但改善结构和性能。由于这些名称含义的不确定性,本章尽量避免使用这类术语。

图 9-12 显示了根据最新调查数据(Davidsen and Krogstie 2010)得出的维护费用的大致分布。该项研究将维护成本分布与 1980—2005 年的一些早期研究进行了比较,发现维护成本的分布在这 30 年间变化很小。虽然没有更多的最新数据,但这表明这种分布仍然很大程度上是正确的。修复系统缺陷不是最昂贵的维护活动。通过系统演化适应新环境以及新的或发生变化的需求通常会花费大部分的维护工作量。

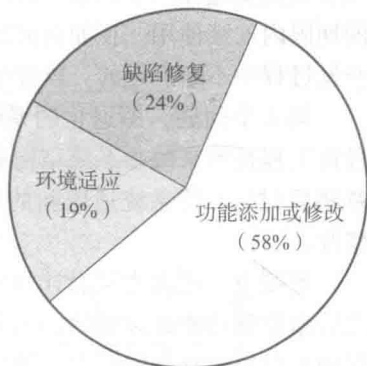


图 9-12 维护工作量分布

通常在系统投入使用之后增加功能,较之在开发期间实现相同的功能代价要高得多,主要原因如下有以下 4 点。

1. 团队稳定性。系统移交之后通常要解散团队,把人员分配到其他新项目中。负责系统维护的新团队或个人既不了解该系统,也不了解系统设计决策的背景,这样在对系统进行变更之前就要花费很多精力来理解现有系统。

2. 糟糕的开发实践。系统的维护合同一般是独立于系统开发合同的。维护合同是与另外的公司签署的,而不是与原开发者签署的。这个因素连同缺乏团队稳定性因素一起,使开发团队缺乏动力去写维护性好的软件。如果一个开发团队为减少工作量而走捷径,即使意味着以后软件的改动会更加困难,他们也认为值得去做。

3. 人员技术水平。维护人员一般都缺乏经验,而且不熟悉应用领域。软件工程人员对维护活动没有什么好印象,通常认为维护不需要太多技术,不如做系统开发那么光彩,所以通常是分配最低级的职员去做。此外,旧的系统可能是用已经淘汰的程序语言写成的,维护人员可能没有多少使用这些语言开发的经验,必须经过学习方能胜任维护工作。

4. 程序年限和结构。随着程序不断的变更,其结构受到了破坏。结果是,随着程序年龄的增加,它们变得越来越不容易理解和变更。此外,许多遗留系统没有使用现代化的软件工程技术来开发。这些系统的结构在设计之初就没有规划好,而且系统开发通常只注重效率优化而很少考虑其易理解性。这些老系统的文档要么没有要么就是不完整,还有可能缺乏一致性。旧的系统也没有采用配置管理,因此在进行系统变更时,常常要在寻找系统构件的合适版本上浪费时间。



文档

系统文档可以通过为维护者提供关于系统的结构和组织的信息以及系统向用户提供的特征来帮助维护过程。虽然敏捷方法的支持者认为代码应该是首要的文档,更高层的设计模型以及关于依赖和约束的信息可以使理解代码以及修改代码变得更容易。

<http://software-engineering-book.com/web/documentation/> (web chapter)

前三个问题的存在是因为很多组织仍然把系统开发和系统维护看作独立的活动。维护被

视为第二等的活动,而且没有动力为减少系统变更开销而投资。要想彻底解决这个问题,首先必须接受这样一个观点:系统很少有一个确定的生存周期,而是以某种形态在一个不确定的期限内连续使用。正如前面所说的那样,应当将系统看成是贯穿于它生命周期的在连续的开发过程中不断演化的。软件维护应该和新软件开发有同等的地位。

第4个问题,即退化的系统结构问题,在某种程度上讲是最容易解决的问题。可以通过再工程技术来改善系统结构和可理解性。如果适当的话,可以通过体系结构的转换(在本章稍后讨论)使系统适应新的硬件。重构能够提升系统代码的质量并且可以使之更加容易修改。

原则上,投入力量设计和实现一个系统来降低未来变更的成本几乎总是合算的。在交付之后添加新功能是昂贵的,因为必须花时间学习系统并分析所提出的变更的影响。在开发过程中所做的对软件进行组织使之更容易理解和修改的工作可以降低演化成本。好的软件工程技术,例如精确的规格说明、测试先行的开发、面向对象开发、配置管理的使用都有助于降低维护成本。

这些为通过提高系统可维护性上的投资来获得整个生命周期的成本节省提供了原则上的论据,然而不幸的是这些论据无法通过真实数据来证实。收集数据很昂贵,数据的价值难以判断。因此,绝大多数公司都认为收集和分析软件工程数据不值得。

实际上,大多数企业都不太愿意在软件开发上花费更多的精力来降低长期维护成本。他们之所以不情愿主要是因为以下两个原因。

1. 公司制订季度或年度支出计划,并鼓励管理人员减少短期成本。投资可维护性会导致短期成本上涨,这部分是可衡量的。然而,与此同时,长期收益却很难衡量,所以企业不愿意将钱花在未来回报未知的事情上。

2. 开发人员通常不负责维护他们所开发的系统。因此,他们很少想到通过一些额外的工来降低维护成本,因为他们不会从中获益。

解决这个问题的唯一方法是集成开发和维护,从而使最初的开发团队在整个软件生命周期中始终对软件负责。对于软件产品和诸如亚马逊等开发和维护自己的软件的公司来说,这是可能的(O'Hanlon 2006)。然而,对于由软件公司为客户开发的定制化软件来说,这是不可能发生的。

9.3.1 维护预测

维护预测关注评估软件系统可能需要的变更,并识别系统中变更的代价有可能最高的部分。如果理解这些,那么就可以针对最有可能发生变更的软件构件进行特别的设计,从而使其具有更好的适应性。还可以投入力量去改进这些构件,以减少其生命周期维护成本。同样,应当试着去估计在给定时间内的系统的维护成本。图9-13说明了对这些不同方面的预测和相关问题。

预测变更请求的数量需要了解系统和外部环境之间的关系。许多系统与外部环境之间存在着复杂的关系,对环境所做的改变不可避免地导致系统变更的发生。要对系统和系统的环境之间的关系做出判断,应该评估以下几点。

1. 系统接口的数量和复杂性。接口越多、越复杂,接口变更请求就越有可能作为新的需求被提出来。

2. 自身具有易变性的系统需求的数量。如第4章中所讨论的,那些反映组织政策和流程

的需求比那些基于稳定的领域特性的需求更容易变动。

3. 系统被使用时所处的业务过程。业务过程在演化的时候,就会产生系统变更请求。使用系统的业务过程越多,要求系统变更的请求就会越多。

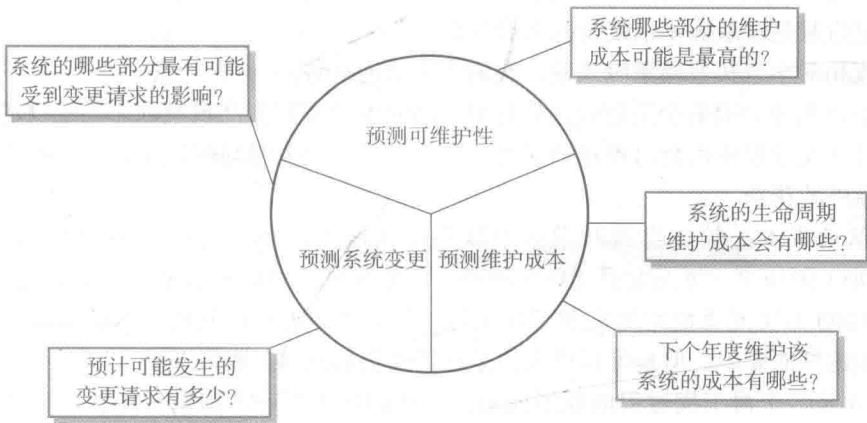


图 9-13 维护预测

很多年以来,研究人员一直在研究程序复杂性和可维护性(Banker et al. 1993, Coleman et al. 1994, Kozlov et al. 2008)之间的关系。这些研究结论都在意料之中:系统或构件越复杂,其维护费用就越高。复杂性度量在识别那些维护费用特别高的个别程序构件时特别有用。因此,用比较简单的构件去代替特别复杂的系统构件是划算的。

当系统已经投入服务的时候,可以使用过程数据来帮助预测可维护性。对可维护性评估有用的过程度量包括以下4个。

1. 请求纠正性维护的数量。如果失败报告的数量在增加,这可能暗示着在维护过程期间有更多的错误引入程序之中了,这样可能会导致系统可维护性的下降。
2. 影响分析所需的平均时间。它反映了受到变更请求影响的程序构件数量。如果这个时间在增加,就暗示着越来越多的构件受到影响,可维护性正在下降。
3. 实现一个变更请求的平均时间。这不同于影响分析所需的时间,尽管它们之间可能存在关联性。它所涉及的活动是对系统及其文档进行变更,而不是只评估哪些构件受到影响。如果实现变更的时间在增加,这可能预示着可维护性在下降。
4. 突出的变更请求的数量。如果这种变更请求数量随着时间在增加,可能意味着可维护性在下降。

我们根据变更请求的预测信息和系统可维护性的预测信息来预测维护费用。多数管理者是将这些信息和自己的直觉、经验相结合来进行成本估计的。成本估计的COCOMO 2模型指出:软件维护工作量是基于理解现有代码的工作量和开发新代码的工作量来估计的。详细讨论见第23章。

9.3.2 软件再工程

正如前面章节所讨论的,系统演化过程包括去理解要变更的程序,然后去实现这些变更。但是,对于很多系统,特别是遗留的老系统,它们是很难理解和进行变更的。这些程序可能最初牺牲了一些可理解性来换取在性能上或空间利用上的改善。此外,随着时间的推

移, 最初的程序结构经过一系列的变更后已被破坏了。

为了使得遗留系统的维护问题变得更简单, 可以再工程这些系统以增强它们的结构性和可理解性。再工程包括对系统重新建立文档、重构系统体系结构、用一种更先进的程序设计语言转换系统、修改和更新系统的数据结构和系统的数据取值。一般来讲, 软件的功能不会改变, 也应当避免对系统体系结构的大的改动。

再工程相对于直接替换系统来说, 有两个重要的优势。

1. 较小的风险。对某个关键业务软件的重新开发是要冒很高风险的。系统描述中会发生错误, 而且开发过程中也会出现种种问题。在引入新软件上时间的拖延将意味着商业上的损失且招致额外的花费。

2. 较低的成本。较之重新开发一个软件所用成本, 再工程的成本要小得多。Ulrich (Ulrich 1990) 引用了一个商业系统作为例子, 该系统再实现的成本高达 5 000 万美元。最后系统仅用 1200 万美元就成功地实现了再工程。如果运用先进的软件技术, 重新实现的花费与上面的相比可能要少, 但是可以肯定的是它仍然会超过再工程的花费。

图 9-14 是一个再工程过程的通用模型。过程的输入是一个遗留程序, 输出是同一个程序的一个已改进和重新构造的版本。这个再工程过程中的活动如下。

1. 源代码转换。使用转换工具, 将程序从旧的程序设计语言转换到相同语言的一个比较新的版本或另一种语言。

2. 逆向工程。对程序进行分析, 并从中抽取信息来记录它的组织结构和功能。这个过程通常是全自动完成的。

3. 程序结构改进。对程序的控制结构进行分析和修改, 使它更容易阅读和理解。

4. 程序模块化。程序的相关部分被收集在一起, 在一定程度上消除冗余。在某些情况下, 这个阶段可能包括体系结构的重构 (例如, 一个系统本来使用几个不同的数据存储器, 结果被要求使用一个单独的存储器)。

5. 数据再工程。改变程序处理的数据以反映程序变更。这可能意味着重新定义数据库模式和将已存在的数据库向新的结构转变。需要经常清理数据, 包括查找和改正错误、删除冗余记录, 等等。

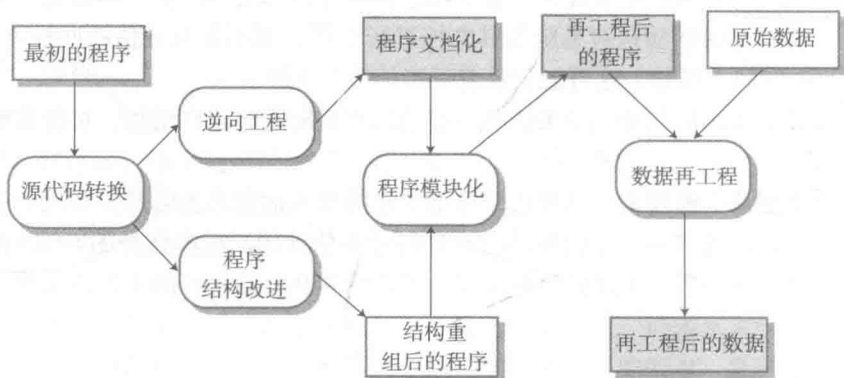


图 9-14 再工程过程

程序再工程可能不需要图 9-11 中显示的所有步骤, 如果仍使用应用程序的编程语言, 源代码就没有必要做转换; 如果再工程完全依赖于自动化工具, 那么通过逆向工程来恢复文

档就没有必要了。数据再工程只有在系统再工程期间程序中数据结构发生了改变时才是需要的。

正如第 19 章所讨论的,为了使再工程系统和新软件互操作,我们可能需要开发适配器构件。这样就隐藏了软件系统的最初的接口,呈现出新的、较好的结构化接口,可以被其他构件使用。对于开发大型的可复用构件来说,遗留系统的封装是一项很重要的技术。

再工程的成本很显然依赖于所做的工作的程度。图 9-15 给出了再工程所可能使用方法的一个谱系。成本从左到右逐渐增长,所以源代码转换是最便宜的选项,而再工程加上一部分体系结构迁移是费用最高的选项。

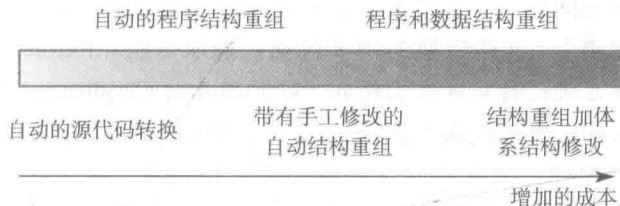


图 9-15 再工程方法

软件再工程的缺点是,系统经过再工程后能改善的范围受到一定的限制。举例来说,它不可能把面向功能的系统转换为面向对象的系统;主要体系结构的变更或对系统数据管理的重新组织不能自动地执行,因此需要额外的高成本;虽然再工程能改善可维护性,但经过再工程的系统不可能像用现代软件工程方法开发的新系统一样好维护。

9.3.3 软件重构

重构(refactoring)是提升程序以减缓其由于更改而退化的过程。它意味着通过修改程序来改进程序结构性,降低程序复杂性,让程序变得更加易于理解。重构有时被认为局限于面向对象的开发,但是其原理可以被任何开发方法所使用。当重构一个程序时,不应该增加其功能,而应该注重程序的改进。因此,可以把重构看成是“预防性的维护”,以此来减少未来变更产生的问题。

重构是类似于极限编程这样的敏捷方法的一个固有部分,因为这些方法都是应对程序变更的。因此,程序的质量容易退化得很快。所以敏捷开发者经常重构它们的程序来避免这样的退化。对于敏捷方法中对回归测试的强调,降低了由于重构引进新错误的风险。引入的任何错误都是可检测的,因为之前成功的测试这时会失败。然而,重构不依赖于其他的“敏捷活动”,并且能够被任何方法用于开发。

尽管再工程和重构都是要将软件变得更加容易理解和变更,但它们并不是同一回事。再工程发生在系统已经维护了一段时间并且维护费用不断上升的情况下,通过使用自动化工具来处理并再工程一个遗留系统,产生一个更具可维护性的新系统。重构是一个连续不断的改进过程,它贯穿于开发和演化的整个过程。重构是要避免导致成本上升和维护困难的结构以及代码的退化问题。

Fowler 等人(Fowler et al. 1999)表示,存在一些固定模式的情况(Fowler 等称之为“坏味道”),其中程序的代码能够被改进。这些能够通过重构被改进的情况包括如下这些。

1. 冗余代码。在程序的不同地方相似的代码可能重复出现很多次。这种情况可以删除它并用一个方法或要求的功能去实现。

2. 长方法。如果方法太长了,那么这个方法应当被重新设计成几个较短的方法。

3. 选择语句。这种情况常常牵扯到重复,因为选择语句 switch 依靠的是同一个值的不同类型。选择语句可能分散在程序的各个地方。在面向对象语言中,常常可以通过多态性来实现同一个事情。

4. 数据聚集。当同样的一组数据项(类中的域,方法中的参数)在程序的不同地方重复出现时,数据聚集就出现了。这通常可以通过用一个对象封装所有数据来解决。

5. 假设的一般性。即开发者为了以后万一使用到在程序中包含了一般性。这通常可以简单地删除掉。

Fowler 在他的书和网站中,也给出了一些基本的重构转换,这些重构可以被单独或一起使用来处理“坏味道”。这些转换的例子包括:抽取方法(Extract Method),移除重复的部分并创建一个新方法;合并条件表达式(Consolidate Conditional Expression),用一个条件测试替换一系列条件测试;提升方法(Pull Up Method),用父类中的一个方法将各个子类中的类似方法替换掉。交互式开发环境,例如 Eclipse,在它的编辑器中提供了对重构的支持。这样使得发现程序中的相互依赖的部分更为容易,这些部分在实现重构时需要修改。

重构在程序开发期间实现,是一项有效降低程序长期维护成本的途径。然而,当你需要维护一个其结构已经明显退化的程序时,仅仅通过重构代码是远远不够的。可能还要考虑对设计重构,这可能是一个花费更高和更加困难的问题。设计重构包括识别相关设计模式(详见第7章),并用实现这些实际模式的代码替换原先的代码(Kerievsky 2004)。

要点

- 软件开发与演化应当是一个集成的、完整的、增量式的过程,它可以用螺旋模型表示。
- 对于定制系统来说,软件维护的费用一般超过了软件开发的费用。
- 软件演化过程是由变更请求驱动的,包括变更影响分析、版本规划和变更实现。
- 遗留系统是旧的软件系统,它使用过时的软件和硬件技术开发,但对于企业仍然有用。
- 应该对遗留系统的业务价值、应用软件的质量以及应用软件的环境进行评估,然后才能决定是否更换、转换和维护系统。
- 软件维护有3种基本类型:修复软件中的缺陷、使软件适应不同运行环境,以及向系统中添加或修改功能。
- 软件再工程的目标是改善系统结构和文档,使其更容易理解和变更。
- 重构做出一些小的程序修改同时保持原有功能不变,它可以看作是一种预防性的维护。

阅读推荐

《Working Effectively with Legacy Code》这本书提供了关于处理遗留系统遇到的问题和困难的可靠实用的建议。(M. Feathers, 2004, John Wiley & Sons)

《The Economics of Software Maintenance in the 21st Century》这篇文章是软件维护的一个简要概述,并且对维护的开销做了详细的介绍。Jones 讨论了影响维护开销的因素并

且发现 75% 的软件工作量都和维护活动有关。(C. Jones, 2006) <http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf>

《You Can't Be Agile in Maintenance?》这篇博文讨论了敏捷开发的方法是否适合维护, 并且讨论了 XP 所建议的技术中有哪些是有效的。(J. Bird, 2011) <http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html>

《Software Reengineering and Testing Considerations》是印度大型软件公司发布的关于维护问题非常出色的一本白皮书。(Y. Kumar and Dipti, 2012) <http://www.infosys.com/engineering-services/white-papers/Documents/software-re-engineering-processes.pdf>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap9/>

支持视频的链接: <http://software-engineering-book.com/videos/implementation-and-evolution/>

练习

- 9.1 为什么在现实环境中使用的软件系统必须进行变更, 否则就会逐渐失去其作用?
- 9.2 从图 9-4 中, 你会发现影响分析是软件演化过程中的一个重要子过程。利用图示, 请说出哪些特性需要在变更影响分析中考虑。
- 9.3 为什么遗留系统应当被视为社会技术系统而不仅仅是一个简单的使用过时技术的软件系统?
- 9.4 什么情况下一个组织会决定废弃一个被评估为高质量和商业价值的系统?
- 9.5 对遗留系统演化的策略选择是什么? 什么时候你通常会替换整个或部分系统而不是继续进行软件维护?
- 9.6 为什么在支持软件上出现的问题会使组织不得不替换遗留系统?
- 9.7 你是公司的一名软件项目经理, 专门从事离岸石油业的软件开发, 你现在需要发现影响公司软件系统的维护性的因素。你该如何设置一个程序去分析维护的过程以及提出并度量软件的维护指标?
- 9.8 简要描述 3 种不同类型的软件维护。为什么有时候很难区分它们?
- 9.9 解释软件再工程和重构的区别。
- 9.10 软件工程师是否具有一种专业的责任感尽力使所开发的代码的维护开销较低, 即使雇主并没有明确要求这么做?

参考文献

Banker, R. D., S. M. Datar, C. F. Kemerer, and D. Zweig. 1993. "Software Complexity and Maintenance Costs." *Comm. ACM* 36 (11): 81-94. doi:10.1145/163359.163375.

Coleman, D., D. Ash, B. Lowther, and P. Oman. 1994. "Using Metrics to Evaluate Software System Maintainability." *IEEE Computer* 27 (8): 44-49. doi:10.1109/2.303623.

Davidson, M. G., and J. Krogstie. 2010. "A Longitudinal Study of Development and Maintenance." *Information and Software Technology* 52 (7): 707-719. doi:10.1016/j.infsof.2010.03.003.

Erlikh, L. 2000. "Leveraging Legacy System Dollars for E-Business." *IT Professional* 2 (3 (May/June 2000)): 17-23. doi:10.1109/6294.846201.

- Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.
- Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.
- Jones, T. C. 2006. "The Economics of Software Maintenance in the 21st Century." www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf.
- Kerievsky, J. 2004. *Refactoring to Patterns*. Boston: Addison-Wesley.
- Kozlov, D., J. Koskinen, M. Sakkinen, and J. Markkula. 2008. "Assessing Maintainability Change over Multiple Software Releases." *J. of Software Maintenance and Evolution* 20 (1): 31–58. doi:10.1002/smr.361.
- Lientz, B. P., and E. B. Swanson. 1980. *Software Maintenance Management*. Reading, MA: Addison-Wesley.
- Mitchell, R. M. 2012. "COBOL on the Mainframe: Does It Have a Future?" *Computerworld US*. <http://features.techworld.com/applications/3344704/cobol-on-the-mainframe-does-it-have-a-future/>
- O'Hanlon, C. 2006. "A Conversation with Werner Vogels." *ACM Queue* 4 (4): 14–22. doi:10.1145/1142055.1142065.
- Rajlich, V. T., and K. H. Bennett. 2000. "A Staged Model for the Software Life Cycle." *IEEE Computer* 33 (7): 66–71. doi:10.1109/2.869374.
- Ulrich, W. M. 1990. "The Evolutionary Growth of Software Reengineering and the Decade Ahead." *American Programmer* 3 (10): 14–20.
- Warren, I. (ed.). 1998. *The Renaissance of Legacy Systems*. London: Springer.

系统可依赖性和信息安全

由于软件系统现在已经成为我们生活各个方面的组成部分，所以我们在软件工程中面临的最大挑战是确保我们可以信任这些系统。信任一个系统，就是说我们必须有信心当需要的时候系统会是可用的并且按照期望的方式运行。系统必须保证信息安全，从而使我们的计算机或数据不会受到威胁，并且系统必须在发生失效或受到网络攻击的时候迅速恢复。因此，本书的这一部分关注软件系统可依赖性和信息安全这些重要话题。

第 10 章介绍了可依赖性和信息安全的基本概念，即可靠性、可用性、安全性、信息安全、韧性，解释了为什么构造信息安全并且可依赖的系统不仅仅是一个技术问题，介绍了用于创建可靠的以及信息安全的系统的冗余和多样性这两个基本的机制。每个可依赖性属性在后续的章节中详细介绍。

第 11 章关注可靠性和可用性，介绍了这些属性如何被刻画为失效或停机的概率，讨论了一些面向容错系统体系结构的体系结构模式，以及可以用于减少系统中的缺陷数量的开发技术。在最后一节中，解释了系统的可靠性如何测试和度量。

越来越多的系统现在都是安全关键系统，其中系统失效会危及到人。第 12 章关注可以用于开发这些安全关键系统的安全工程和技术。解释了为什么安全性是一个比可靠性更广的概念，并且讨论了得出系统安全需求的方法。还解释了为什么针对安全关键性系统工程定义和描述的过程很重要，并且描述了软件安全案例——一种用于说明一个系统为什么安全的结构化文档。

对系统信息安全的威胁是当今社会面临的一个主要问题，为此本部分用了两章来介绍这个话题。第 13 章关注应用信息安全工程——在各个软件系统中实现信息安全的方法。解释了信息安全和其他可依赖性属性之间的关系，并且介绍了信息安全需求工程、信息安全系统的设计、信息安全系统测试。

第 14 章是新的一章，针对更广的韧性话题。韧性系统可以在系统的一些部分失效或者遭受网络攻击时继续提供它的重要服务。本章解释了网络安全的基础，并讨论了韧性是如何使用冗余和多样性以及通过人员授权和技术机制来实现的。最后，讨论了有助于提高系统韧性的系统和软件设计问题。

可依赖系统

目标

本章的目标是介绍软件可依赖性的概念，以及开发一个可依赖的系统所涉及的内容。读完本章后，你将：

- 了解可依赖性和信息安全性对软件系统的重要性；
- 了解可依赖性的 5 个重要维度，即可用性、可靠性、安全性、信息安全性和可恢复性；
- 了解社会技术系统的概念，以及为什么需要将其视为一个整体系统而不是简单的软件系统；
- 了解冗余性和多样性在实现可依赖的系统和过程中起到的重要作用；
- 树立使用形式化方法构建可依赖的系统的意识。

随着计算机系统已经深入到我们的业务和个人生活当中，系统与软件的失效所带来的问题也在增加。一个电子商务公司的服务软件的失效有可能导致收入上较大的损失，并有可能殃及这家公司的顾客。一个嵌入汽车上的控制软件发生错误可能会导致昂贵的召回和维修费用，并且在最坏的情况下，可能成为导致意外事故的因素之一。公司里的电脑如果被恶意代码感染，则需要昂贵的清除操作以找出问题所在，并可能导致损失或敏感信息的毁坏。

因为软件密集型系统对于政府、公司和个人都如此重要，所以广泛使用的软件必须是可信赖的。这种软件应该在需要时是可用的，应该能够正确地工作且不会出现不受欢迎的副作用，例如未授权的信息泄露。简而言之，我们应该感到软件系统是足够可靠的。

可依赖性 (dependability) 这个概念是由 Jean-Claude Laprie 在 1995 年提出来的，目的是覆盖系统的可用性、可靠性、安全性和信息安全性。他提出的这一理论在接下来的数年中被不断讨论和修改，并最终在 2004 年正式发表于论文当中 (Avizienis et al. 2004)。正如 10.1 节所讨论的，这些属性是交织在一起的，所以用一个词来覆盖它们是有意义的。

系统的可依赖性通常比它们的具体功能更加重要，这主要是因为以下几个方面的原因。

1. 系统失效影响到很多人。很多系统中包含了一些很少被使用的功能。如果这些功能被从系统中移除掉，那么只有一小部分人会受影响。而影响系统可用性的系统失效潜在地影响所有使用系统的用户。系统失效可能意味着正常的业务无法进行。

2. 用户经常拒绝不可靠、不安全或信息安全有问题的系统。如果用户发现一个系统不可靠或者信息安全有问题，他们就会拒绝使用它。更有甚者，他们还有可能拒绝购买和使用来自同一个公司的其他产品。他们不希望重复他们在一个不可依赖性的系统上的糟糕经历。

3. 系统失效的代价可能是巨大的。对于某些应用，例如反应堆控制系统或者飞机导航系统，系统失效的代价要比一般控制系统失效的代价大好几个数量级。诸如电源之类的核心控制系统的失效会造成不可估量的损失。

4. 可依赖性差的系统可能导致信息丢失。数据收集和维持的成本很高，有时甚至比处理

它的计算机系统还昂贵。恢复丢失的或被损坏的数据所需要的花费往往非常高。

然而，一个有用的系统并不一定总是具有很高的可依赖性。例如，用来写这本书的文字处理软件就并不是很可靠，它有时候会突然失去响应或重新启动。即便如此，由于它的功能比较完善，我也可以容忍这种偶尔发生的意外情况。但是，由于我不是绝对信赖这个系统，所以我会经常地保存文件并且额外进行一些备份操作。正因为我针对可依赖性的缺失做了额外的对应处理，因此当系统崩溃时我可以把损失降到最低。



关键性系统

某些系统属于“关键性系统”类别，其系统故障可能导致人身伤害、破坏环境或巨大的经济损失。关键性系统的例子包括医疗中的嵌入式系统设备，如胰岛素泵（安全关键），航天器导航系统（关键任务）和在线转账系统（关键业务）。

关键系统开发非常昂贵。它们不仅必须被开发出来，以便故障极少发生，它们还必须包括恢复机制，以便在故障发生时和之后使用。

<http://software-engineering-book.com/web/critical-systems/>

构建可依赖的软件是构建可依赖的系统工程中最重要的一部分。正如第 10 章中所讨论的，软件通常是一个大的系统中的一部分。它在一个运行环境下执行，这个环境中包括有能令软件在其上执行的硬件、软件的使用者，还有组织过程或业务过程。因此当设计一个可依赖的系统时，我们需要考虑以下这些方面。

1. 硬件失效。系统硬件失效有可能是源自设计上的失误，也可能源自构件加工制造中的问题，或者是硬件构件已达到它们的使用年限了。

2. 软件失效。系统软件问题可能是由于规格说明、设计和实现中的错误。

3. 操作失效。系统的操作人员未能按照预期正确地使用系统。因为硬件和软件正在逐渐变得可靠，所以操作上的失效就成为最大的一个引起系统失效的原因了。

这些失效可能是相互关联的。硬件构件的失效可能需要系统操作员必须处理不可预料的情况，因而增加了额外的负载，这使得他们精神紧张——人在这种紧张情况下更容易出错。这又会导致软件失效，软件失效又意味着需要操作员做更多的工作，神经更加紧张，等等。

由此看来，对于开发可靠的、软件密集型系统的设计者来说尤其重要的是，要有整体的社会技术系统观，不能只注意系统的某个局部。如果系统的硬件、软件和操作过程的设计是孤立考虑的，一个部分的设计者没有注意到其他部分的潜在弱点，那么就有可能在系统的各构件接口处出错。

10.1 可依赖性属性

我们都知道计算机系统失效是怎么一回事。在没有明显原因的情况下，计算机系统有时会崩溃或者出错。在计算机上运行的程序没有像预料的那样执行，有时可能损坏系统中的数据。我们已经习惯了这些失效，很少有人完全信任所使用的计算机。

计算机系统的可依赖性是可信赖度（trustworthiness）的性能指标。可信赖度表现为用户对系统的信任程度，系统是否能按照他们预期的那样操作以及系统是否会在正常使

用中失效。以数值量化可依赖性没有多大意义,相反,我们将其分为这样几个层次可能会更好:“不可依赖”“非常可依赖”“极度可依赖”。

如图 10-1 所示,可依赖性包含了如下 5 个维度。

1. 可用性。一般来讲,系统的可用性(availability)是指系统在任何时间都能运行并能够提供有用服务的可能性。

2. 可靠性。一般来讲,系统的可靠性(reliability)是系统在给定的时段内能正确提供用户希望的服务的可能性。

3. 安全性。一般来讲,系统的安全性(safety)是判断系统将会对人和系统的环境造成伤害的可能性。

4. 信息安全性。一般来讲,系统的信息安全性(security)是判断系统能抵抗意外的或蓄意的入侵的可能性。

5. 韧性。系统的韧性(resilience)是指当出现一些干扰性事件(比如,设备故障或者恶意攻击)的时候,系统保持其关键服务继续正常运行的可能性。这一特性是在最近才被加入到最初由 Laprie 所提出的可依赖性属性之中的。

图 10-1 所示的可依赖性属性都很复杂,它们可以分解为一些其他更简单的属性。例如,信息安全性包括完整性(保证系统程序和数据没有损坏)、机密性(保证信息只能由得到授权的人访问)。可靠性包括正确性(保证系统服务按照所定义的那样提供)、精确性(保证信息按照需要的细节层次传送)和及时性(保证信息能在规定的时间内传送到)。

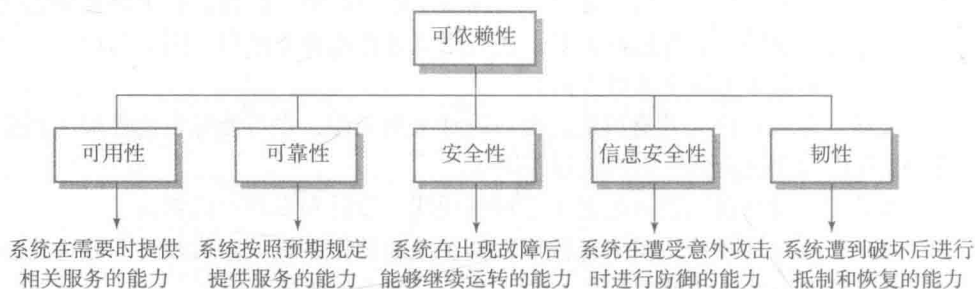


图 10-1 可依赖性属性原则

当然,这些可依赖性属性不是对所有系统都是同样重要的。在第 1 章介绍的胰岛素泵系统中,最为重要的属性是可用性(在需要的时候必须可用)和安全性(决不能传输危险剂量的胰岛素)。在这里,信息安全性就不是个问题了,因为泵不维护机密信息,也不在网络环境下工作,所以不会受到恶意攻击。对于野外气象系统,可用性和可靠性是最重要的特性,因为维修的费用是非常高的。对于病人信息系统,信息安全性和韧性是特别重要的,因为其维护的都是敏感的私人数据,并且这些数据随时都可以作为患者咨询的重要依据。

还有以下一些系统属性与 5 个可依赖性属性密切相关,而且会影响系统的可依赖性。

1. 可维修性。系统失效是不可避免的,但是如果系统可以很快修复的话,系统失效而导致的崩溃就可以尽量避免。为了做到这一点,就必须能诊断问题,找到失效的构件并加以修复。当系统的组织能够获取软件源代码并有修改代码的能力时,软件的可维修性会提高。开源软件在这一点上更容易一些,但是对构件的复用会使得做到这一点更困难。

2. 可维护性。在系统使用过程中新需求会不断出现,改变它使之保持有用性并不断适应新需求就变得相当重要。可维护软件能够以低成本的修改来应对新需求,而且在修改过程中

引入新错误的可能性比较小。

3. 容错。这可以看成是可用性的一部分，反映为在多大程度上系统的设计能避免用户输入错误并在输入有错时能够容忍错误的存在。当用户发生错误时，系统应该尽量检测这些错误，而且要么能够自动修改错误要么请求用户重新输入数据。

系统可依赖性的概念是作为一个全面的属性来介绍的，因为它的可用性、信息安全性、可靠性、安全性、韧性这些属性都是紧密联系的。安全的系统运行通常需要系统是可行的并且可靠地运行。一个系统可能由于一个入侵者破坏了它的数据而变得不可靠。对一个系统的拒绝服务攻击其目的是损坏系统的可用性。如果一个系统被病毒感染了，用户就不可能再对它的可靠性和安全性有信心，因为病毒可能会改变系统的行为。

为了开发一个可靠的软件，你需要做到：

1. 避免在软件规格说明和开发过程中引入意外的错误。
2. 设计检验和确认 (V&V) 过程，使之能够有效地发现影响系统可靠性的残余错误。
3. 设计容错机制，确保出现错误时系统仍能继续工作。
4. 设计保护机制防范能够损坏系统可用性和信息安全性的外部攻击。
5. 正确地配置和部署系统及它的支持软件，以提供良好的运行环境。
6. 提高系统能力以识别外部网络攻击并抵抗这些攻击。
7. 设计恢复机制来保证系统失效或遭受攻击时不会丢失重要数据。

容错的需求意味着可靠的系统必须包含冗余代码用以监视系统本身，探测错误状态，并且在失效发生之前从错误中恢复。这会影响到系统的表现，当每次系统执行时都要做额外检查。因此，设计者总是要在系统性能和可依赖性两者之间做出折中。可能因为系统运行速度变慢而弃用系统检查。然而，随之而来的风险是，由于一些缺陷没有被发现而导致发生一些失效。

因为额外的设计、实现和确认成本，加强系统可依赖性会大大增加开发成本。尤其是对于需要过度可依赖性的系统，如安全性要求极高的控制系统，验证和确认 (V&V) 的成本会非常高。除了需要检验系统是否符合其需求定义外，还需要向外部管理者证明系统是安全的。例如，航空器系统必须向外部管理者，比如国家航空管理局，证明系统是安全的，影响到飞行器的安全的灾难性系统失效发生的概率是极低的。

图 10-2 给出了成本和对可依赖性渐增改善之间的关系。如果软件可依赖性不是很高，可以通过更好的软件工程方法用较低的代价显著提高软件质量。然而，如果你已经采用了很好的措施，改进的花费将高很多而得到的效益却比较低。还有一个问题是测试你的软件并证明你的软件是可靠的也绝非易事。解决这个问题需要运行多次测试并查看一定数量发生的错误。随着软件可依赖性变得更高，你将会看到越来越少的失效。结果，需要更多的测试并猜测你的软件还有多少问题。测试是非常昂贵的，这急剧增加了高可用性系统的成本。

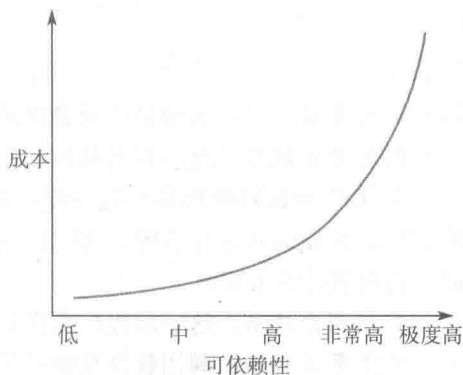


图 10-2 成本 - 可依赖性曲线

10.2 社会技术系统

在计算机系统中，软件和硬件是相互依赖的。没有硬件，软件系统就是抽象的，它仅

仅是人的知识与理念的一种表示；没有软件，硬件只不过是一套没活力的电子设备。但是，如果将二者放到一起构成一个系统，就创造了一个可以进行复杂计算并给环境提供结果的机器。

这说明了系统的基本特性之一——系统，大于其各部分相加之和。系统的一些特性只有在各部分集成并在一起运行时才会展现出来。因此软件工程不是一个孤立的活動，而是一个更一般的系统工程过程中的固有组成部分。软件系统不是独立的系统，更多的是含有人、社会或组织目的更广泛系统的基本成分（见第 19 章）。

例如，野外气象系统软件控制一个气象站里的工具。它与其他软件系统通信，并且是国内及国际气象预报系统的一部分。和软硬件一样，这些系统包括天气预报过程、操作系统及分析结果的人员。系统还包括依靠它向个人、政府、企业等提供天气预报的组织。这类更广泛的系统有时也被称作“社会技术系统（sociotechnical system）”。他们同时包含了非技术因素，如人、流程、规章等，以及技术因素，如计算机、软件和其他设备。系统的可依赖性取决于社会技术系统中的每一个元素——硬件、软件、人和组织。

社会技术系统实在过于复杂，以至于我们无法从整体上直接理解它。因此，图 10-3 通过分层的方法展现了它的内部结构。以下 7 个层次组成了社会技术系统栈。



图 10-3 社会技术系统栈

1. 设备层。这一层由硬件设备组成，其中一些部分可能是计算机。
2. 操作系统层。这一层与硬件层交互，为系统中较高层的软件层提供一套公共工具。
3. 通信和数据管理层。这一层扩展了操作系统工具，提供了允许与更多扩展功能交互的接口，比如访问远程系统，访问系统数据库等。因为这一层通常在应用程序和操作系统之间，有时被称为中间件。
4. 应用系统层。这一层提供所需的特定应用功能，可能包含多个不同的应用程序。
5. 业务过程层。利用软件系统的组织业务过程在这一层定义及制定。
6. 组织层。这一层包含更高层的战略流程以及业务规则、政策和使用系统时要遵循的标准。
7. 社会层。这一层定义了支配系统运作的社会法律和规章。

注意，到这里并没有独立的“软件层”，因为它在社会技术系统中的每一层中都或多或少地扮演着重要的角色。对于嵌入式系统，起到控制作用的是设备；而对于操作系统和应用，起到控制作用的则是软件。因此，业务流程、组织和社会都依赖于互联网（软件）以及其他全球性的软件系统。

原则上说,大多数交互是在相邻层进行的,每一层对上一层隐藏下一层的细节。但实际上并不总是这样,有时也存在跨层次间的意外交互,从而给整个系统带来问题。例如,假设规定个人信息访问的法律发生了变化,这一点属于社会层。它引发了新的组织程序和业务过程的变化。但是,应用程序系统本身可能不能提供所需层次的细节,因此也要在通信和数据管理层做出相应的调整。

顾全整个系统,而不是简单地从孤立的软件角度来看,这对于考虑软件的信息安全性和可靠性是非常必要的。软件的失效很少会对它自身造成严重的后果,因为软件是无形的存在,即便受到损坏,也可以轻而易举地修复。但是,当这些软件的失效波及系统的其他部分时,它们会影响软件的物理环境以及人的环境。此时,失效的后果更加严重。重要的数据可能会丢失或损坏。人们不得不做额外的工作从失效中恢复,比如设备的损毁,数据的丢失和损坏,或者秘密泄露导致的未知后果。

所以,当设计一个对信息安全性和可靠性有要求的软件时,必须从系统全局观察,了解软件失效带给系统中其他部件的后果。此外,系统中的其他部件亦有可能造成或防止软件失效,以及协助从软件失效中恢复。

最重要的目标是确保当软件发生失效时,不会引入整个系统的失效。因此必须检查软件如何与其直接环境进行交互来确保以下两点:

1. 软件失效尽可能包含在系统栈的封闭层中,不会对相邻层的操作产生严重影响。软件失效不应该导致系统失效。

2. 理解系统栈的非软件层错误和失效是如何影响软件的,同时考虑怎样将检测点置入软件中,从而协助检测失效,以及如何提供对失效恢复的支持。

因为软件固有的柔性,许多预想不到的问题同时也就摆在了软件工程师面前。例如,雷达的位置选择之后可能发现雷达图像有扭曲现象,这时通过移动雷达重新定位已经不现实了,解决方案可能就是通过软件来增强雷达图像处理能力,消除扭曲现象。这可能又降低了软件的性能以至于性能变得无法接受。此时该问题可能被看成是“软件失效”,而实际上这是系统整体设计过程上的失效。

软件工程师必须面对在不准许提高硬件费用的前提下增强软件能力的问题,这种情况并不少见。许多所谓的“软件失效”不是软件固有问题的结果。这种失效是尝试通过改变软件以满足已变更的系统工程需求的结果。一个说明这种失效的最好的例子是丹佛机场行李系统的失效(Swartz 1996),在那儿,遇到了许多硬件设备上的限制,需要修改控制软件来弥补。

10.2.1 规章与守约

现在世界上几乎所有经济组织的普遍做法是,私有公司提供商品和服务,并从中赚取利润。我们有一个竞争环境,这些公司在成本、质量、交货时间等方面进行竞争。然而,为了确保其公民的安全,大多数政府限制私有公司的自由,强制他们必须遵守某些标准,以确保他们的产品是安全和放心的。因此,公司不能以更低的价格销售产品,因为这样做会导致产品的安全性降低。

各国政府在不同领域都制定了一套规则和条例来确保产品的安全性和信息安全。他们还建立了监管组织,确保在某一地区的公司提供的产品符合这些规则。监管组织拥有很高的权力,如果有公司违反规定,他们可以进行罚款,甚至监禁董事。他们还扮演着许可证授权方

的角色（例如，在航空业和核工业），为新使用的系统颁发许可证。因此，飞机制造商必须向购买方所在国的监管组织出示适航证书。

为了获取认证，开发安全关键系统的公司必须制定一个具有泛用性的安全条例（在第13章中讨论），表明其已经遵守了既定的规则和法规，并使监管方相信系统可以安全地运行。开发这样的安全条例是非常昂贵的，甚至用于开发安全条例的费用会与开发系统本身的费用相当。

规章和守约（遵守规则）适用于整个社会技术系统，而不仅仅是适用于该系统的软件部分。例如，核工业中的调节器会时刻检测是否存在反应堆过热的异常，以确保它不会将放射性物质释放到环境中。使监管组织相信这种安全措施有效的证据是软件保护系统、用于监测反应堆堆芯的操作过程，以及包裹任何放射性物质的外壳结构的完整性。

这些元素中的每一个都必须有自己的安全条例。因此，保护系统必须有一个安全条例，证明软件将正常运行，并按预期关闭核反应堆。总体情况还必须表明，如果软件保护系统发生故障，则有其他不依赖软件的安全机制被调用。

10.3 冗余和多样性

任何系统中的构件失效都是不可避免的。人们犯错误，未被发现的软件错误导致不良行为，硬件也会损坏。我们使用一系列策略来减少人为失效的数量，例如，在预计使用寿命结束之前更换硬件构件，并使用静态分析工具检查软件。但是，我们无法确定这些将会消除构件失效。因此，我们应该设计系统，以使单个构件失效不会导致整个系统失效。

实现和提高可依赖性的策略依赖于冗余和多样性。冗余性意味着系统中包含多余的能力可以在系统失效的时候发挥作用。多样性意味着冗余的系统构件是属于不同种类的，这样它们就难以完全一样的方式失效。

我们使用冗余和多样性来增加我们日常生活中的可依赖性。通常，为保证我们的居所安全我们通常使用几道锁（冗余），并且，通常我们使用不同种类的锁（多样性）。这意味着如果一个人入侵者找到破坏其中一个锁的方法，他们还必须找到不同的方法来破坏其他的锁才能进入。如日常工作一样，我们应该完全备份我们电脑中的内容，并保持数据的多个冗余拷贝。为避免磁盘失效的问题，备份应该在单独的、不同的外部设备上保存。

设计用来增强可依赖性的软件系统可能包括提供与系统中的其他构件相同功能的冗余构件。这些构件在主构件失效的时候被有选择地加入系统。如果这些冗余构件是多样的（比如，与其他构件不同），一个普通的缺陷不会引起一个有备份构件的系统的失效。冗余性也可以通过包含附加的检查代码来提供，这些代码可能并不是直接对系统功能有用。可以在一些缺陷导致失效之前探测到这些代码。它们可以激发恢复机制工作以保证系统继续运行。

在可用性要求高的系统中，冗余的服务常常被使用。这些服务在指定的服务失效的时候自动运行。有时候，为保证攻击不能够暴露一个常见的弱点，这些服务可能属于不同的类型并可能在不同的操作系统上运行。使用不同的操作系统是软件多样性和冗余的例子之一，在这里相似的功能以不同的方式提供。（第12章将详细讨论软件多样性。）

多样性和冗余可以同样用于可依赖性软件开发过程的设计。可依赖性开发过程要避免对系统引入错误。在可依赖性开发过程中，例如软件确认之类的活动不依赖于单一的工具或技术。这增强了软件的可依赖性，因为它降低了过程失效的机会，也就是由于在软件开发过程中人的错误导致的软件错误。



阿丽亚娜 5 型火箭爆炸

1996 年，欧洲航天局的阿丽亚娜 5 型火箭在其首次飞行开始 37 秒后发生爆炸。事件是由于软件系统失效造成的。火箭上有个软件备份系统，但这个系统并不具有多样性，所以备份计算机也以同样的方式失效了。火箭和其卫星的有效载荷全部被摧毁。

<http://software-engineering-book.com/web/ariane/>

比如说，确认活动可能将程序测试、手工程序检查以及静态分析作为缺陷查找技术。这些技术中的任何一种都可能找到其他方法所遗漏的错误。另外，不同的组员可能负责一些活动（比如，程序检查）。人们用何种方法处理工作取决于他们的个性、经验和受教育程度，所以这种冗余性提供了对系统不同的视角。

如第 11 章介绍的那样，使用软件的多样性和冗余会对软件引入错误。多样性和冗余使系统更加复杂并且通常难以理解。不仅仅因为需要编写更多的代码并检查它们，还需要有附加的功能加入到系统中来以探测构件的失效并切换控制到候选构件。这些额外的复杂性使程序员更容易犯错误，且系统检查人员更难以检查到错误。

因此，一些工程师认为最好避免软件冗余和多样性。他们的观点是，设计软件要越简单越好，可以使用极为严格的软件验证和确认（V & V）过程（Parnas, van Schouwen, and Shu 1990）。更多的花费可以用在验证和确认上，因为节省了开发冗余系统构件的花费。

两个方法都被用在商业上的安全关键系统中。比如，空客 340 飞行控制系统硬件和软件既多样又冗余。波音 777 的飞行控制软件是基于冗余的硬件，但每个计算机都运行同样的软件，这个软件是得到极度严格的确认的。波音 777 飞行控制系统把注意力集中在简单性而不是冗余性上。这两个飞行控制系统都是可依赖的，所以很明显，多样性和简单方法对于实现可依赖性来说都是成功的。



可依赖的操作过程

本章讨论可依赖的开发过程，但是对于系统的可依赖性还有一个同样重要的方面，那就是系统的运行过程。在设计这些系统的操作过程中，需要考虑人的因素，并总是在心中牢记人在使用系统时是很容易出错的。设计一个可依赖的过程应该避免人为错误，当错误形成时，软件应该能检测到此错误并允许使用者去改正错误。

<http://software-engineering-book.com/web/human-error/>

10.4 可依赖的过程

可依赖软件过程是用来开发可依赖软件的软件过程。在可依赖过程上的投资的理由是，一个好的软件过程可以使所交付的软件包含更少的错误，并因此有更少的执行时失效。公司使用了可依赖过程，保证这个过程能得到正确执行和文档化，且适当的开发技术用在了关键系统开发中。图 10-4 显示了一些可依赖软件过程的特性。

对外部管理者来说，应用可依赖过程的证据能令他们相信在软件开发中已经使用了最有效的软件工程实践。系统开发人员也将定期拿出一个过程模型以及过程被遵循的证据给管理者看。要使管理者相信所有参与者都一致地使用这个过程，并且该过程可以被用在不同的开发项目中。这意味着过程必须被明确地定义并且是可重现的。

1. 明确定义的过程就是用已定义的过程模型来驱动软件生产过程。在过程中必须被收集的数据说明开发团队遵循了过程模型中定义的过程。

2. 可重复的过程是一个不依赖于个别解释和判断的过程。此过程可以在各个项目中重复使用，并且在不同的团队人员中使用，不论谁在开发团队中。这对于关键系统是特别重要的，因为这样的系统通常有一个很长的开发周期，并且在这个周期中开发小组的成员会经常发生变动。

可依赖过程利用冗余和多样性来获得可靠性。可依赖过程经常包含相同目标的不同活动。比如，程序审查和测试的目标是发现程序中的错误。这些方法是相互补充的，同时采用两种技术可能比只使用一种技术能发现更多的错误。

过程特点	描 述
可审计	过程应该是能够被外部人员理解的，外部人员会检查过程标准是否得到遵守，并给出过程改善的若干意见和建议
多样性	过程应该包括冗余和多样性验证和确认活动
可文档化	过程应该有一个定义好的过程模型，定义过程中的活动以及要在活动中产生的各种文档
鲁棒性	过程应该能够从单个过程活动的失效中恢复
标准化	该有一个全面的覆盖软件开发和文档化的一组软件开发标准

图 10-4 可依赖过程的属性

这些活动用在可依赖性过程中显然依赖于所开发的软件的种类。然而总的来说，这些活动应该相互协调以防止在系统中引入新的错误，探测并移除错误，同时维护过程本身的信息。在一个可依赖的过程中可能包括的活动有以下这些。

1. 需求评审，尽可能地检查需求的完备性和一致性。
2. 需求管理，保证需求的变更是受控制的，并且所有的受到变更影响的程序员都能够了解所提出的需求变更的影响。
3. 形式化规格说明，创建并且分析软件的数学模型（在 10.5 节讨论了形式化规格说明的益处）。可能它最重要的益处是它对系统需求施加一个非常详细的分析。这个分析本身可以用来发现可能在需求评审阶段被忽略的需求问题。
4. 系统建模，也就是用一组图形化的模型清晰地记录软件设计，并且需求和这些模型之间的联系也被清晰地记录下来。如果使用了模型驱动的工程方法（见第 5 章），代码可能通过这些模型自动生成。
5. 设计和程序审查，让不同的人分别检查系统不同的描述部分。通常是依据普遍性的设计和编程错误清单进行审查的。常见设计和编程错误的清单可能对审查过程很有帮助。
6. 静态分析，就是对程序的源代码进行自动检查。查找那些可能指示程序错误或疏忽的异常（第 12 章将讨论静态分析）。
7. 测试规划和管理，即设计一系列全面的系统测试。必须很小心地管理测试过程，证明这些测试已经覆盖了系统需求并且在测试过程中得到正确的应用。

如同聚焦系统开发和测试的过程活动,我们还必须有完善定义的质量管理和变更管理过程。尽管一家公司的可依赖性过程中的特殊活动可能与另一家公司不同,对有效的质量管理和变更管理的需要是一致的。

质量管理过程(将在第24章讨论)建立了一系列过程和产品标准。它们还包括捕捉过程信息的活动以证明这些标准被遵循了。比如,可能存在一个标准,它被定义来执行程序审查。审查小组的组长负责记录过程以说明审查标准一直是得到遵循的。

在第25章讨论的变更管理是关于管理系统的变更,确保一个接受的变更能够真正实现,并肯定所计划的发行版本中包含了这些计划中的变更。一个通常出现的软件问题是在一个系统中包含了一个错误的构件。这可以导致一种情况的发生,即一个执行系统中包含在开发过程中未经检查的构件。配置管理过程必须定义为变更管理过程的一部分,以保证此类事件不会发生。

敏捷方法被越来越多地使用,研究者和实践者都在考虑如何在可依赖软件开发中应用敏捷方法(Trimble 2012)。大多数开发关键软件系统的公司有基于自身的有计划的开发过程,他们都不愿对开发过程进行彻底的改变。但是这些公司认识到了敏捷开发的价值,并在探索如何把自身的开发过程变得敏捷。

验证可依赖软件需要过程和产品的文档化,同时还需要预先进行需求分析来发现可能危害到系统安全的需求和需求冲突。正式的变更分析对评估变更对系统安全性和完整性的影响至关重要。这些要求与敏捷开发中共同开发需求和系统的一般方法以及最小化文档相冲突。

尽管大多数敏捷开发过程使用非正式的且无须文档化的过程,但这不是敏捷的根本需求。敏捷过程包含例如迭代开发,测试优先开发和在用户开发团队中的参与的技术。只要一个团队遵循这个过程并记录了他们的活动,敏捷方法就可以被使用。为了支持这个观点,已经提出了各种包含可依赖系统开发需求的改进的敏捷方法建议(Douglass 2013),把敏捷开发和基于计划开发中的最合适的技术连接了起来。

10.5 形式化方法与可依赖性

30多年来,研究者提出了使用形式化方法来进行软件开发。形式化方法是基于数学的软件开发方法,其中需要定义软件的形式化模型。接着可以形式化地分析这些模型来找到错误和不一致的地方,从而证明程序与模型一致。或对这个模型应用一系列保持正确性的变换来生成程序。Abril(Abril 2009)声称使用形式化方法可以产生“无错误的系统”,虽然他很小心地限定了他在声明中的意思。

在一份调查中,Woodcock等人(Woodcock et al. 2009)讨论了已经成功应用形式化方法的工业应用。这些应用包括列车控制系统(Badeau and Amelot 2005),刷卡系统(Hall and Chapman 2002)和飞行控制系统(Miller et al. 2005)。形式化方法是静态验证中使用的工具的基础,例如微软使用的驱动程序验证系统(Ball et al. 2006)。

在计算机科学发展的早期阶段提出了使用数学形式化方法进行软件开发。这个想法是指形式化规格说明和程序可以独立开发,然后开发一个数学证明来表明程序及其规格说明是一致的。最初,证明是人工开发的,但这是一个漫长而昂贵的过程。人们很快发现人工证明应用于非常小的系统。现在很多自动定理证明软件支持程序验证,这意味着更大的系统是可以被证明的。然而,开发定理证明者的证明义务是一个困难且专业的任务,因此形式化验证没有被广泛使用。

避免单独的证明活动的可替代方法是基于精化的开发。这里，系统的形式化规格说明通过一系列保持正确性的变换被改进，从而生成软件。因为这些是可靠的变换，所以可以确信生成的程序与其形式化规格说明一致。这是巴黎地铁系统软件开发中使用的方法 (Badeau and Amelot 2005)，它使用一种所谓的 B 语言 (Abrial 2010)，该语言旨在支持对规格说明的精化。

基于模型检查的形式化方法 (Jhala and Majumdar 2009) 已被用于大量的系统中 (Bochot et al. 2009; Calinescu and Kwiatkowska 2009)。这些系统依赖于构建或生成系统的形式化状态模型，并且使用模型检查器来确认一直维持模型的属性 (例如安全属性)。模型检查程序详细地分析形式化规格说明，并且检查程序会报告系统属性通过模型被满足，或者展示出不满足的示例。如果模型可以从程序中自动或系统地生成，这意味着程序中的错误。(在第 12 章讨论安全关键系统中的模型检查)。



形式化规格说明技术

形式化规格说明可以采用两种基本方法表达：一是用系统界面模型 (代数规格说明)；二是用系统状态模型。读者可以下载一个额外的网上关于此话题的一章内容，其中给出了两种方法的例子。在那一章中包含胰岛素泵系统的一部分的形式化规格说明。

<http://software-engineering-book.com/web/formal-methods/> (在线章节)

软件工程的形式化方法对于发现或避免以下两类软件表示的错误是有效的。

1. 规格说明和设计中的错误和遗漏。开发和分析软件的形式化模型的过程中，可能揭示软件需求中的错误和遗漏。如果从源代码自动或系统地生成模型，则使用模型检查的分析可以发现可能发生的不期望的状态，例如，并发系统中的死锁。

2. 规格说明和程序间的不一致。如果使用精化方法，可以避免出现开发者使软件与规格说明不一致的错误。程序验证可以发现程序与其规格说明之间的不一致。

所有形式化方法都开始于一个数学系统模型，把它作为系统的规格说明。为了建立这个模型，需要将自然的语言、图和表格等表述的用户需求翻译成其他形式化定义语义的机器语言。形式化规格说明是对系统应该做什么的一个无二义的描述。

形式化规格说明对于设计的证明和软件的实现不仅仅是必不可少的，还是定义系统最精确的方式，减少了误解的可能。许多形式化方法的支持者相信，创建形式化规格说明是值得的，即使没有精化或程序证明。另外，构造一个形式化的规格说明需要一个细致的需求分析，而这又是一个发现需求存在的问题的有效方法。在一个自然语言规格说明中，错误可能被不精确的语言所掩盖，而在使用了形式化规格说明的系统中不会出现这样的问题。

开发形式化规格说明并将其用于一个形式化开发过程的优势包括下面这些。

1. 当详细地开发一个形式化规格说明的时候，对系统需求获得了深度细致的理解。改正早期发现的需求问题通常比在晚一些的开发过程中修改它们代价相对要小很多。

2. 因为规格说明是用一种带有形式定义语义的语言来表达的，因此可以自动地分析它来发现其中的不一致和不完整之处。

3. 如果你使用一个方法 (比如 B 方法)，你可以使用一系列正确性保持变换将形式化规

格说明转换成程序。这样其结果程序可以保证与规格说明是一致的。

4. 程序测试的花费可能会降低, 因为你已经根据规格说明验证了程序。例如, 在开发巴黎地铁系统软件时, 精细化意味着不需要软件构件测试, 只需要进行系统测试。

Woodcock 调查 (Woodcock et al. 2009) 发现, 使用形式化方法的人在交付的软件中产生的错误更少, 且在可比较的开发项目中, 软件开发所需的成本和时间更少。在需要监管组织认证的安全关键系统中使用形式化方法有显著的好处。所生成的文档是系统安全条例的重要组成部分 (参见第 12 章)。

尽管有这些优点, 形式化方法在实际软件开发中的影响有限, 即使对于关键系统也是如此。Woodcock 研究了 25 年中使用形式化方法的 62 个项目。即使考虑到了使用这些技术但没有报告其使用的项目, 这仍是当时开发的关键系统总数中的一小部分。工业一直不愿采用形式化方法, 原因如下。

1. 提出问题的人和领域专家不能理解形式化规格说明, 所以他们不能检查形式化规格说明是否能表达他们的需求。软件工程师理解形式化规格说明, 但是却可能不了解应用领域, 所以他们同样不能确定形式化规格说明是否是系统需求的精确反映。

2. 对创建一个形式化规格说明的花费是很容易量化的, 但是评估因使用它所带来的成本节余就要困难得多。结果, 管理者不愿意冒风险来采用形式化方法。总的来说, 他们不相信成功的报告, 因为这些报告来自开发者提倡形式化方法的特殊项目。

3. 多数的软件工程师没有受过形式化规格说明语言的训练。这样, 他们也就不愿意在开发过程中提议使用这个方法。

4. 很难将现有方法扩展到一个很大的系统中。当使用形式化方法时, 最通常的做法是只对关键核心软件进行形式规格说明, 而不是对整个系统使用此方法。

5. 对形式化方法的工具支持有限, 可用的工具往往开源且难以使用。对于商业化工具提供商来说市场太小。

6. 形式化方法与程序逐步开发的敏捷开发不相容。但这不是一个主要问题, 因为大多数关键系统仍然是使用基于计划的方法开发的。

早期倡导形式化开发的 Parnas 批评了当前的形式化方法, 并声称这些方法从一个根本错误的前提开始 (Parnas 2010)。他认为, 这些方法将不会被接受, 直到它们被彻底简化, 这将需要不同类型的数学作为基础。本书的观点是, 这并不意味着形式化方法被常规地用于关键系统工程, 除非能够清楚地证明与其他软件工程方法相比, 其采用和使用是合算的。

要点

- 系统可依赖性很重要, 因为关键计算机系统的故障可能导致巨大的经济损失, 严重的信息丢失, 物理损坏或对人类生命产生威胁。
- 计算机系统的可依赖性反映用户对系统的信任程度的系统属性。可依赖性的最重要的维度是可用性、可靠性、安全性、信息安全性和韧性。
- 社会技术系统包括计算机硬件、软件和人, 并且位于组织内, 旨在支持组织或业务目标。
- 如果要最小化系统中的故障, 使用可依赖的、可重复的过程是必要的。该过程应包括从需求定义到系统实现的所有阶段的验证和确认活动。
- 在硬件、软件过程和软件系统中使用冗余和多样性对于可靠系统的开发是至关重要。

要的。

- 形式化方法使用系统的形式化模型作为开发的基础，有助于减少系统的规格说明和实现中错误的数量。然而，出于对这种方法的成本效益的担心，形式化方法在工业中的使用有限。

阅读推荐

《Basic Concepts and Taxonomy of Dependable and Secure Computing》这项工作提出了一些可依赖性概念的深入讨论，这些概念是由负责开发它们的一些先驱者编写的。（A. Avizienis, J. -C. Laprie, B. Randell 和 C. Landwehr. IEEE Transactions on Dependable and Secure Computing, 1 (1), 2004）<http://dx.doi.org/10.1109/TDSC.2004.2>

《Formal Methods: Practice and Experience》是一个对工业界中形式化方法的使用的一个很好的调查，包含一些使用形式化方法的项目的规格说明。作者总结了使用这些形式化方法的障碍。（J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Computing Surveys, 41 (1) January 2009）<http://dx.doi.org/10.1145/1592434.1592436>

《The LSCITS Socio-technical Systems Handbook》这本手册以简单易懂的方式介绍社会技术系统，并提供了关于社会技术主题的更详细的论文。（2012）<http://archive.cs.st-andrews.ac.uk/STSE-Handbook/>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap10/>

支持视频的链接: <http://software-engineering-book.com/videos/critical-systems/>

练习

- 10.1 给出软件可依赖性对于大多数社会技术系统十分重要的 6 个理由。
- 10.2 什么是系统可依赖性中最重要的方面？
- 10.3 举例解释为什么在开发可依赖的系统时需将其视为社会技术系统，而不能简单地将其看作技术软件和硬件系统。
- 10.4 举两个由复杂社会技术系统支持的政府职能的例子，并解释为什么在可预见的未来这些职能不能完全自动化。
- 10.5 解释冗余和多样性的不同之处。
- 10.6 为什么假设使用可依赖的过程可以创建可依赖的软件是合理的？
- 10.7 举两个可以归入可依赖过程的多样性、冗余的活动的例子。
- 10.8 给出基于软件多样性的不同版本的系统可能以类似的方式失效的两个原因。
- 10.9 你是一名负责开发小型、安全关键的列车控制系统的工程师，该系统必须具有安全性和信息安全性。你建议在这个系统的开发中使用形式化方法，但你的经理对这种方法持怀疑态度。撰写一份报告，展现形式化方法的好处，并提出一个供他们在本项目中使用的条例。
- 10.10 有人认为监管的必要性阻碍了创新，并且监管组织强制使用在其他系统上使用过的旧的系统开发方法。讨论你是否认为这是真的，监管组织对于应该使用什么方法强加自己的观点是否可取。

参考文献

- Abrial, J. R. 2009. "Faultless Systems: Yes We Can." *IEEE Computer* 42 (9): 30–36. doi:10.1109/MC.2009.283.
- . 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.
- Avizienis, A., J. C. Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Trans. on Dependable and Secure Computing* 1 (1): 11–33. doi:10.1109/TDSC.2004.2.
- Badeau, F., and A. Amelot. 2005. "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL." In *Proc. ZB 2005: Formal Specification and Development in Z and B*. Guildford, UK: Springer. doi:10.1007/11415787_20.
- Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. 2006. "Thorough Static Analysis of Device Drivers." In *Proc. EuroSys 2006*. Leuven, Belgium. doi:10.1145/1218063.1217943.
- Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.
- Calinescu, R. C., and M. Z. Kwiatkowska. 2009. "Using Quantitative Analysis to Implement Autonomous IT Systems." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 100–10. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE.2009.5070512.
- Douglass, B. 2013. "Agile Analysis Practices for Safety-Critical Software Development." <http://www.ibm.com/developerworks/rational/library/agile-analysis-practices-safety-critical-development/>.
- Hall, A., and R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18–25. doi:10.1109/52.976937.
- Jhala, R., and R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4), Article 21. doi:1145/1592434.1592438.
- Miller, S. P., E. A. Anderson, L. G. Wagner, M. W. Whalen, and M. P. E. Heimdahl. 2005. "Formal Verification of Flight Critical Software." In *Proc. AIAA Guidance, Navigation and Control Conference*. San Francisco. doi:10.2514/6.2005-6431.
- Parnas, D. 2010. "Really Rethinking Formal Methods." *IEEE Computer* 43 (1): 28–34. doi:10.1109/MC.2010.22.
- Parnas, D., J. van Schouwen, and P. K. Shu. 1990. "Evaluation of Safety-Critical Software." *Comm. ACM* 33 (6): 636–651. doi:10.1145/78973.78974.
- Swartz, A. J. 1996. "Airport 95: Automated Baggage System?" *ACM Software Engineering Notes* 21 (2): 79–83. doi:10.1145/227531.227544.
- Trimble, J. 2012. "Agile Development Methods for Space Operations." In *SpaceOps 2012*. Stockholm. doi:10.2514/6.2012-1264554.
- Woodcock, J., P. G. Larsen, J. Bicarregui, and J. Fitzgerald. 2009. "Formal Methods: Practice and Experience." *Computing Surveys* 41 (4): 1–36. doi:10.1145/1592434.1592436.

可靠性工程

目标

本章的目标是介绍如何刻画、实现和度量软件的可靠性。阅读完本章后，你将：

- 理解软件可靠性和软件可用性的区别；
- 了解用于可靠性规格说明的度量，以及如何用这些度量去刻画可度量的可靠性需求；
- 理解如何使用不同的体系结构风格去实现可靠的、容错的系统体系结构；
- 了解可靠性软件工程中好的编程实践；
- 理解如何用统计测试去衡量一个软件系统的可靠性。

我们的业务和个人生活的所有方面几乎都依赖于软件系统。这意味着当我们需要软件时，我们希望它是可用的。这种依赖可能是在周末或假期的清晨或深夜，软件必须天天运行，一年到头每一天都必须如此。我们希望软件运行时不会崩溃或者发生失效，同时能保护我们的数据和个人信息。我们要能够信任我们所使用的软件，这意味着软件必须是可靠的。

在过去的 20 年中，软件工程技术的使用、更好的程序设计语言、有效的质量管理已经使软件可靠性得到了显著提高。然而，系统失效依然出现，这些故障影响系统可用性或导致了不正确结果的产生。在软件扮演着关键性角色的情景下，也许在航天飞机里或者作为国家关键基础设施的一部分，专用的可靠性工程技术可能用于实现我们需要的高等级的软件可靠性和可用性。

然而，在谈到系统可靠性时很容易感到困惑，因为对系统故障和失效不同的人有不同的说法。Brian Randell 是软件可靠性的先驱，他定义了一个基于以下概念的“故障（fault）- 错误（error）- 失效（failure）”模型（Randell 2000），即人为错误导致系统故障，系统故障导致系统错误，系统错误导致系统失效。他准确地定义了人为错误、系统故障、系统错误、系统失效这些术语。

1. 人为错误。所有导致在系统中引入故障的人的行为。例如，在野外气象系统中，程序员可能决定计算下一次传输数据的时间为当前时间加 1 小时。当传输数据的时间在 23:00 和午夜（午夜时间为 00:00，对于 24 小时制）之间时，这种计算就会出错。

2. 系统故障。可以导致系统出错的软件系统特性。在上面的例子中，故障出现在以下代码：变量 `Transmission_time` 加 1。这段代码没有检查该变量的值是否大于或等于 23:00。

3. 系统错误。一个错误的系统状态，能够导致系统行为完全超出系统用户预期。当故障代码被执行时，传输数据时刻的值设置得不正确（应该为 24:xx，而不是 00:xx）。

4. 系统失效。在某一时刻所发生的事件，系统不能提供一个用户所期待的服务。在这个例子中，没有气象数据传出，因为时间是无效的。

系统故障不会必然导致系统错误，并且系统错误不一定导致系统失效。见如下几种情况。

1. 不是程序中所有的代码都会执行。包含故障的代码（比如，初始化一个变量的失败）可能因软件使用的方式等原因永远不会被执行。

2. 错误是短暂的。一个状态变量可能由于故障代码的执行产生一个不正确的值。然而，在这个变量被访问并导致系统失效之前，其他的系统输入可能已经被处理，而此事件会重置这个状态变量为一个有效值。

3. 系统可能拥有故障探测和保护机制。这些机制确保在系统服务被影响之前发现错误行为并进行改正。

故障不一定会引起系统失效的另一个原因是，在实践中，用户会调整行为以避免使用他们知道会产生程序失效的输入。有经验的用户会依据他们的经验避开不可靠的软件特征。例如，在有些人使用的文字处理系统中，会避开使用诸如自动编号的软件功能，因为用户的经验是：自动编号经常会出错。修复不使用的功能中的故障不会改变系统可靠性。

故障、错误和失效之间的差别，帮助我们找出 3 个可以用于改善系统可靠性的辅助方法：

1. 故障避免。在系统的设计和实现过程中使用一些软件开发方法来减少编程故障发生，并在系统投入使用之前发现系统中的故障。更少的故障意味着更少的运行时失效的机会。故障避免技术包括使用强类型的程序设计语言以允许全面的编译器检查和尽量不要使用容易出错的程序设计语言元素，（例如指针）。

2. 故障检测和修复。在部署使用之前进行验证和确认（V&V）来发现和去除程序中的故障。关键性系统需要广泛的验证和确认过程，以便在部署之前发现尽可能多的故障，并且令系统的拥有者和管理者确信系统是可靠的。系统化的测试和调试以及静态分析是故障检测技术的实例。

3. 容错。系统应该设计成具备容错能力的，即系统的故障及无法预期的系统行为在执行时能够得到检测和管理，避免导致系统失效。在所有的系统中都应该包含一种基于内嵌的运行时检测的容错方法。更多的特定容错技术，比如在 11.3 节中介绍的容错系统体系结构的使用，通常来说只用在那些需要非常高级别可用性和可靠性的系统中。

不幸的是，应用错误避免、错误检测和容错技术并非总是成本-效益合算的。发现和清除软件系统中的残余故障的成本呈指数增加（见图 11-1）。当软件越来越可靠时，用于查找越来越少的故障所耗的时间和精力将会越来越多。在某个阶段，这些额外花费是不合理的。

因此，软件开发企业默认他们的软件存在一些残余故障。允许的故障水平依赖于系统类型。软件产品允许相对较高的故障水平，而关键系统通常要求极低的故障密度。

可以接受的故障的基本原则是，当系统失效时，失效造成的损失比在系统发布之前发现和消除它们的开销要少。但是发布包含故障的软件并不仅仅是一个经济问题，同时还要考虑系统失效的社会和政治影响。

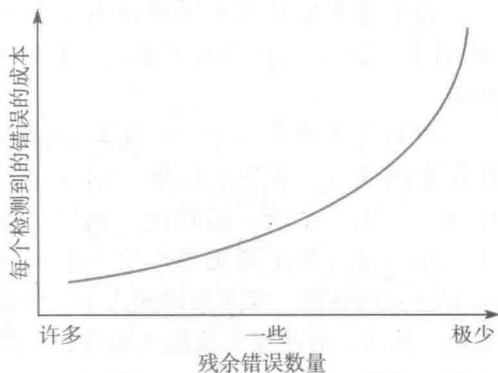


图 11-1 残余故障移除的递增成本

11.1 可用性和可靠性

第10章介绍了系统可靠性和可用性的概念。如果认为系统提供某种服务(例如,交付现金,控制刹车或者连接电话),那么这些服务的可用性是指服务能否启动和运行,可靠性是指服务能否提供正确的结果。系统的可用性和可靠性都可以用概率来表达。如果可用性是0.999,即表示在一段时间里的99.9%都是可用的。如果平均来说每1000次输入中两次输入引起失效,那么其可靠性用发生失效的频率来表示就是0.002。系统可靠性和可用性的更精确的定义如下。

1. 可靠性。系统在一个特定时间、特定环境中针对一个特定目的而执行的无失效操作的可能性。

2. 可用性。系统在一个时刻是可操作的和能执行请求服务的可能性。

系统可靠性不是绝对的,它取决于系统在何地以及如何被使用。举例来说,一个软件系统的可靠性度量是在办公室环境中进行的,环境中绝大多数用户都对软件的操作不感兴趣。他们会按照说明书一板一眼地操作,从不对系统做其他尝试。在一个大学环境中,可靠性可能就大大不同了。在这里,学生探究系统的边界,可能以一种意想不到的方式使用系统。这些都可能造成系统的失效,而在办公室环境中很少发生这种情况。因此,对系统可靠性的看法在各种环境中是不同的。

以上对可靠性的定义是基于系统无失效运行的观点给出的,其中失效是影响系统用户的外部事件。但是失效由什么构成?一个技术性的定义是,失效是一种违反系统规格说明的行为。但是,对于这个定义存在以下两个问题。

1. 软件规格说明通常是不完整的或不正确的,软件工程师用规格说明来解释系统该如何表现。因为他们不是这些领域的专家,他们实现的行为可能不是用户所期待的。软件行为可能是符合规格说明的,但对用户而言是失效的。

2. 没有人希望系统开发人员阅读软件规格说明文档。因此,用户可能期望软件应该以一种和规格说明完全不同的方式表现出来。

因此,不能客观地定义失效。系统失效是由系统用户判断的。这就是用户总是对系统可靠性有着不相同的感觉的原因。

为了理解为什么不同的环境中可靠性表现不同,我们需要把系统考虑成一个输入输出映射。

图11-2说明了一个软件系统是输入到输出集合的映射。程序响应单一输入或者输入序列,产生一个或一组输出。例如,给定一个URL输入,Web浏览器产生一个相应的显示Web页的输出。大多数的输入不会引起系统失效。然而,有些输入或输入组合,如图11-2中用带阴影的椭圆 I_e 所示,引起系统失效或产生错误输出。程序的可靠性取决于作为引发错误输出的输入集合的数量。换句话说,这个输入集合导致了错误代码的执行和系统错误的发生。如果在集合 I_e 中的输入被系统频繁使用的代码执行,那么失效就会频繁发生。然而,如

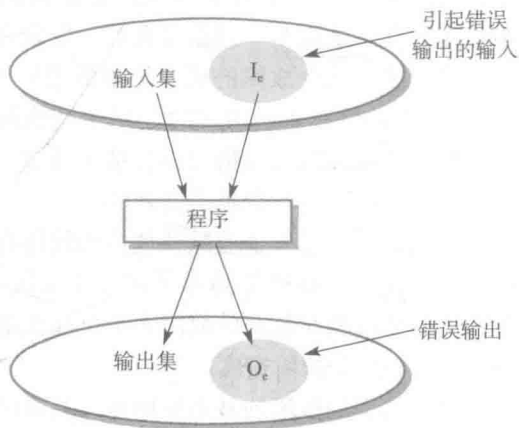


图 11-2 系统的输入/输出映射

果输入集合 I_e 被很少使用的代码执行, 那用户可能几乎不会遇到失效。

对某个用户而言影响系统可靠性的故障对另一个用户而言可能就不是。在图 11-3 中错误的输入集对应图 11-2 中标有 I_e 的椭圆部分。用户 2 使用的输入集与这个错误的输入集相交, 用户 2 将会体会到系统失效的滋味。用户 1 和用户 3 从不使用来自这个错误集合中的输入, 对他们来讲, 这个软件看上去总是可靠的。

可用性不仅取决于系统失效的次数, 而且还取决于修复导致失效的故障所需时间。因此, 如果系统 A 每一年失效一次, 系统 B 每个月失效一次, 则 A 系统比 B 系统相对更可靠。然而, 如果系统 A 需要花 6 个小时才能重新启动, 而系统 B 只需 5 分钟就可以重新启动, 那么会认为系统 B 全年而言 (60 分钟宕机时间) 比系统 A (360 分钟宕机时间) 更具有可用性。用户或许更喜欢系统 B。

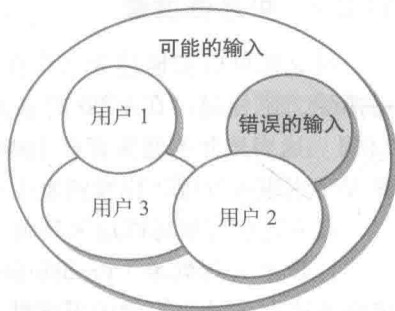


图 11-3 软件使用模式

此外, 不可用系统所造成的混乱不是简单反映在系统的可用性度量上, 可用性度量描述的是不可用时间所占的百分比。系统失效发生的时间点同样很重要。如果一个系统每天在凌晨 3 点到凌晨 4 点不可用, 并不会影响很多用户。然而, 如果同样的系统在上班时间内有 10 分钟不可用, 系统的不可用性可能会对用户造成更大的影响。

可靠性与可用性是紧密联系的, 但是有时候一个比另一个更重要。如果用户期望从系统得到持续的服务, 那么这个系统就有高可用性需求。也就是说无论何时都有需求, 系统都必须是可用的。然而, 如果由于失效而造成的损失不大并且系统可以迅速恢复, 那么这些失效不会严重地影响系统使用者。

电话交换机系统就是一个可用性比可靠性更重要的例子。当拿起听筒的时候, 用户希望听到拨号音, 因此系统有较高的可用性需求。即使一次连接建立起以后系统发生了故障, 故障也可以很快得到修复, 即交换机通常可以重置系统并重新尝试连接。这可能发生得非常之快, 以至于用户甚至没有注意到失效曾经发生。再者, 即便是通话真的被打扰了, 其造成的结果也通常不很严重, 如果这种情况真的发生了, 用户简单重连即可。

11.2 可靠性需求

1993 年 9 月, 一架飞机在风暴中降落波兰华沙机场。开始降落后的 9 秒时间里, 飞机计算机控制的制动系统突然失灵。制动系统并未检测出飞机已经着陆了并认为飞机还在空中。飞机的安全特性中止了逆推力系统的工作, 因为逆推力系统可以降低飞机的速度, 它在空中启动将是危险的。飞机脱离跑道向外冲去, 撞上了泥土堤岸, 引起大火。

对于事故的调查显示, 制动系统软件按其规格说明正常工作了。程序中是没有错误的。然而, 软件的规格说明是不完备的, 它没有考虑到一些极特殊的情况, 恰巧这种极特殊情况发生了。软件正常工作但是系统失效了。

这说明了系统的可依赖性不仅依赖于好的工程过程, 它还需要关注系统需求分析时的一些细节以及用于保证系统可依赖性的软件需求规格说明。可依赖性需求分为以下两种。

1. 功能性需求。定义了系统应该包含的检查和修复措施, 以及防止系统失效和外部攻击的保护性特征。

2. 非功能性需求。定义了系统需要的可靠性和可用性。

正如第 10 章所述，系统的总体可靠性取决于硬件可靠性、软件可靠性，以及操作员的可靠性。系统软件必须考虑到这一点。同时要包括补偿软件失效的需求，可能还要有相关的可靠性需求，以帮助解决检测和恢复硬件失效和操作员错误。

11.2.1 可靠性度量

可靠性可以被描述为系统在一个指定的运行环境中使用时发生失效的概率。如果你愿意接受，比如说，在 1000 次事务处理中有 1 次是失效的，那么你可以描述失效的概率为 0.001。这当然并不意味着在 1000 次事务处理中你一定会看到 1 次失效。它意味着如果你观察 N 千次事务处理，你看到发生失效的次数应该是在 N 这个数附近。

以下这些度量可以用来刻画可靠性和可用性。

1. 请求失效概率（Probability Of Failure On Demand, POFOD）。使用这个度量定义系统服务请求将导致失效的可能性。所以， $POFOD=0.001$ 意味着当提出一个请求的时候可能发生失效的概率是 $1/1000$ 。

2. 失效发生率（Rate Of Occurrence Of Failure, ROCOF）。这个度量说明在一段时间（比如 1 小时），或在一定的系统执行的次数之内，能够观察到的系统失效的次数。在上面的例子当中，ROCOF 是 $1/1000$ 。ROCOF 的倒数是平均失效间隔时间（Mean Time To Failure, MTTF），其常用在可靠性度量当中。MTTF 是观察到的系统失效时间间隔的平均值。ROCOF 为每小时 2 次失效暗示着平均失效间隔时间是 30 分钟。

3. 可用性（AVAIL）。AVAIL 是当请求某一服务的时候系统可使用的概率。因此，0.999 9 的可用性是指平均来说系统在运行时间的 99.99% 是可用的。图 11-4 展示了实践中不同的可用性等级意味着什么。

可用性	解 释
0.9	系统 90% 的时间是可用的。这意味着 24 小时期间（1440 分钟），系统有 144 分钟不可用
0.99	在 24 小时期间，系统有 14.4 分钟不可用
0.999	在 24 小时期间，系统有 84 秒不可用
0.9999	在 24 小时期间，系统有 8.4 秒不可用——大致每个星期有 1 分钟不可用

图 11-4 可用性规格说明

当请求失效会产生一个严重的系统失效的时候，POFOD 应该用作可靠性度量。这一点与提出请求的频率无关。例如，一个保护系统监控化学反应并在过加热的情况下停止反应进行，这就应该用 POFOD 定义的可靠性描述。总的来说，对保护系统的请求是很少发生的，因为这一系统是在所有的修复措施失败之后的最后一道防线。因此 POFOD 为 0.001（1 000 次请求时有一次是失效的）看起来可能是有风险的，但如果在其生命周期中只有两三次对系统的请求，那么可能永远不会发生系统失效。

当系统处理持续的请求而不是断断续续的请求时，应该使用 ROCOF 度量。例如，在一个处理大量事务的系统当中，你可能要指定 ROCOF 为每天 10 次失效。这意味着你愿意接受平均每天 10 次事务处理不能够成功完成并必须取消的事实。要么，你可以定义 ROCOF 为每 1000 次事务中失效的次数。

如果失效之间的绝对时间很重要的话，可以把可靠性定义为平均失效间隔时间（MTTF）。

例如，如果为一个有很长的事务处理序列的系统（比如计算机辅助设计系统）定义可靠性需求，就应该用 MTTF 这个度量指标。MTTF 应该比一个用户在他的模型上连续工作而未存盘的平均时间要长得多，这将意味着用户无论在哪一个工作阶段中都不太可能由于失效丢失数据。

11.2.2 非功能性可靠性需求

非功能性的可靠性需求是对系统可靠性和可用性的要求做出的规格说明，使用前面讲述的度量之一来描述。多年来定量的可靠性和可用性规格说明被用于安全关键系统中，但是很少用在业务关键系统中。然而，随着更多的企业要求其系统提供 24/7 的服务，让服务在可靠性和可用性期望方面变得精确是有意义的。

定量的可靠性规格说明在很多方面是有用的：

1. 决定可靠性需求级别的过程对弄清利益相关者真正的需求是有帮助的。这个过程帮助利益相关者理解不同类型的系统失效的存在，这也使利益相关者清楚地认识到实现高级别的可靠性要花费大量财力。
2. 它提供了一个用来评估何时停止测试系统的基准。当系统达到了它所需的可靠性级别时就可以停止测试。
3. 定量的可靠性规格说明是一种评估不同设计策略的方式，它的目的是提高系统的可靠性。
4. 如果一个系统在投入使用之前已经通过了监管人员的审批（例如，对飞机飞行安全起至关重要的作用的所有系统都有相关规定），那么证明该系统必需的可靠性目标得到满足对系统认证有着重要意义。



可靠性超标

可靠性超标是指对实际运转的软件所定义的必需的可靠性级别比实际需要的更高。可靠性超标不成比例地增加了开发成本，因为减少故障的开销和可靠性验证的花费随着可靠性的提高按指数增长。

<http://software-engineering-book.com/web/over-specifying-reliability/>

为了避免过多的和不必要的花费，对整个系统而言，指定真正需要的可靠性而不是简单选择一个高级别的可靠性是非常重要的。如果一个系统的某些部分相比其他部分而言更关键，那么对系统不同部分可以有不同的可靠性需求。当刻画可靠性需求时，应该遵守下面 3 条准则。

1. 针对不同类型的失效定义可用性和可靠性需求。造成严重损失的失效发生概率应该不会带来重大损失的失效发生概率低。
2. 针对不同类型的服务定义可用性和可靠性需求。关键系统服务应该有最高的可靠性，但是对非关键性的服务你可能愿意容忍更多的失效。一般情况下，考虑到成本效益，只有最关键的系统服务才会使用定量的可靠性规格说明。
3. 考虑是否真的需要高可靠性。比如，可以使用错误探测机制来检查系统的输出并包含

错误修正过程,这样的话产生此输出的系统可能就不需要高可靠性了。

为解释上面3个观点,考虑能够存取现金和为消费者提供其他服务的银行ATM机的可靠性需求。银行对这种系统有两个关注点:

1. 确保系统按照客户的要求执行服务,同时准确地记录账户数据库中的客户交易。
2. 确保这些系统在需要时是可用的。

然而,对于怎样识别和纠正不正确的账户交易,银行拥有多年的经验。他们使用会计方法检查什么时候出错。大多数的事务如果失败了可以简单地取消掉,从而不会引起银行的损失,而且对于用户造成的不便也很小。运行ATM网络的银行因此接受这样的一个事实:ATM的失效可能意味着很小一部分的交易是不正确的,但是事后再修改这些错误较之为了避免错误交易而支付很高的花费要经济得多。因此,ATM需要绝对可靠性的可能性相对较低。每天出现个别的失效是可以接受的。

对于一家银行来说(以及银行的用户),ATM网络的可用性比个别ATM交易是否失效更重要。可用性的缺乏意味着更多的柜台服务请求、更多的用户不满、更多的修复网络的工程花费,等等。因此,对于基于事务的系统,比如银行业和电子商务系统,可靠性描述的焦点通常在于描述系统的可用性。

为了定义一个ATM网络的可用性,应该识别系统的服务,并且对每一项服务定义其所需的可用性需求,包括:

- 客户的账户数据库服务。
- 单个ATM所提供的每一项服务,比如,“取款”“提供打印凭条”等。

这里,数据库的服务是最严格的,因为这项服务的失效意味着网络中的所有ATM都不能提供服务。因此,应该把这项服务描述为高可用的。在这种情况下,早上7时到晚上11时,一个可以接受的可用性数值(忽略安排好的维护和升级)可能是0.9999。这意味着每周不可用的时间少于1分钟。

对于每一个ATM,总的可用性依赖于机械的可靠性和是否现金用尽。软件问题产生的影响应该比这些问题产生的影响要少。因此,ATM的软件可用性级别相对不高是可以接受的。ATM软件的可用性可描述为0.999,意味着在一天当中一台机器不可用的时间为1~2分钟。这允许ATM软件在一个问题事件中重启。

控制系统的可靠性通常定义为一个命令执行时系统将失效的概率(POFOD)。考虑第1章介绍的胰岛素泵中控制软件的可靠性需求。这个系统每天多次注射胰岛素并且每小时对用户的血糖进行多次监控。

可以将胰岛素泵的失效分为两类。

1. 短暂性软件失效。这是一类可以由用户来修复的失效,比如,可以重启或重新校准机器。对于这种类型的失效,相对较低的POFOD值(假设0.002)是可以接受的。这意味着失效可能在每500个请求中发生1次,大约为每3.5天1次,因为血糖含量每小时检查5次。

2. 永久性软件失效。机器需要由厂家维修。这种类型的失效概率非常低才行。大约一年一次是最低要求,因此,POFOD应该不会高于0.000 02。

然而,胰岛素注射失效不会带来生命危险,所以在这里决定可靠性需求级别的是商业因素而非安全因素。由于用户需要一个非常快的维修和更换服务,所以服务的费用是非常高的。厂商的利益决定了需要修理的永久性失效的数量。

11.2.3 功能性可靠性规格说明

为了确保软件密集型系统具有高度的可靠性和可用性，我们需要同时使用故障避免、故障检测和容错技术。这意味着功能性可靠性需求要求系统提供故障避免、检测和容错功能。

这些功能性可靠性需求应当规定需要检测的故障以及应当采取的措施，以确保这些故障不会导致系统失效。因此，功能性可靠性规格说明涉及分析非功能性需求（如果存在这些需求），评估可靠性存在的风险，以及指定系统如何解决这些风险。

系统中有 4 类功能性可靠性需求。

- 1. 检查需求。这些需求明确对系统输入的检查，以确保不正确的或者超过阈值的输入在系统处理之前被检测到。
- 2. 恢复需求。设置这些需求来帮助系统在一次失效发生之后进行恢复。这些需求一般关注的是维护系统及其数据的拷贝，并定义在失效后如何重新修复系统服务。
- 3. 冗余性需求。这些需求定义系统的冗余特征，保证一个构件的失效不会导致整个系统的失效。在下一章中会详细讨论这一点。
- 4. 过程性需求。这些需求是故障避免需求，以确保好的实践在开发过程中得到使用。所指定的实践应当减少系统中的故障数量。

一些可靠性需求的示例如图 11-5 所示。

RR1: 应定义所有操作员输入的预定义范围，并且系统应检查所有操作员输入是否在该预定范围内。 (检查)
RR2: 患者数据库的副本应保存在不同建筑物内的两个不同的服务器上。(恢复，冗余)
RR3: 应用 N 版本编程实现制动控制系统。(冗余)
RR4: 该系统必须在 Ada 的安全子集中被实现，并使用静态分析来检查。(过程)

图 11-5 功能性可靠性需求示例

得出功能性可靠性需求没有简单的规则可循。在开发关键性系统的组织中，内部通常有一些关于可靠性需求和这些需求怎样在实际中影响系统的可靠性的经验。这些组织可能对某种系统特别在行，比如铁路控制系统，这样可靠性需求就可以在此类系统中复用了。

11.3 容错体系结构

容错机制是一种确保运行时可靠性的方法，它定义了系统发生错误后的运行机制，即使发生软件或硬件故障，也能保证系统正常工作。容错机制可以检测并纠正错误状态，使得故障的发生不会导致系统失效。在安全关键和信息安全关键系统中，以及出错后无法继续运行下去的系统中，尤其需要容错机制。

为了提供容错机制，系统体系结构必须具有一定的冗余性，以及多样化的硬件和软件。例如，飞行器系统就需要容错体系结构，以确保通信系统、关键命令和控制系统在飞行过程中始终可用。

可依赖体系结构的最简单实现是采用重复服务器，即两个或多个服务器完成同样的任务。处理的请求通过一个服务器管理构件路由到一个特定的服务器上。这个服务器管理构件同时还跟踪服务器响应。如果服务器失效，这通常通过无法得到响应而被探测到，有故障的服务器被系统剔除出去。未处理的请求由其他服务器重新接收并处理。

这种重复服务器方法广泛使用在事务处理系统中，很容易维护要处理的事务的多个拷贝。事务处理系统的设计使得数据仅仅在事务正确结束才得到更新，这种处理延迟不会影响

系统的完整性。如果备用服务器用在低优先级任务中,那么这个方法可能是使用硬件的一个很有效的方法。如果主要的服务器发生了问题,它的处理将转移到备用服务器进行,该服务器对此任务给予最高优先级。

重复服务器提供冗余性,但是通常不提供多样性。硬件通常是一样的并且运行相同版本的软件。因此它们可以应对集中在一台机器上的硬件失效和软件失效。但是它们不能应对导致所有版本软件同时失效的软件设计问题。为应对软件设计的失效,一个系统必须拥有多样性的软件和硬件。

Torres-Pomales 调查了一系列软件容错技术 (Torres-Pomales 2000), Pullum (Pullum 2001) 描述了不同类型的容错体系结构。下文将描述在容错系统中通常使用的 3 种体系结构模式。

11.3.1 保护性系统

保护性系统是一种与其他系统相关联的专用系统。它通常是对一些过程的控制系统,比如化学制造过程或者一个设备控制系统(如无人驾驶火车系统)。保护性系统的例子可能是一个火车上探测火车是否正在通过一个红色信号灯的系統。如果是,并且没有迹象显示火车控制系统正在减速,那么保护性系统会要求系统制动使其停止。保护性系统独立地监控系统的环境,如果传感器指示控制系统没有正确工作,那么保护性系统被激活并且关闭设备的运行。

图 11-6 说明了保护性系统与控制系统的关系。保护性系统监控受控设备及其环境。如果探测出一个問題,它发出命令使执行器关闭系统或者触发其他保护机制(比如打开一个压力释放阀门)。注意:这里有两个传感器,一个用于正常的系統监控,而另一个专门供保护性系统使用。在传感器失效时,备用机制允许保护性系统继续运作。系统中可能还要有冗余的执行器。

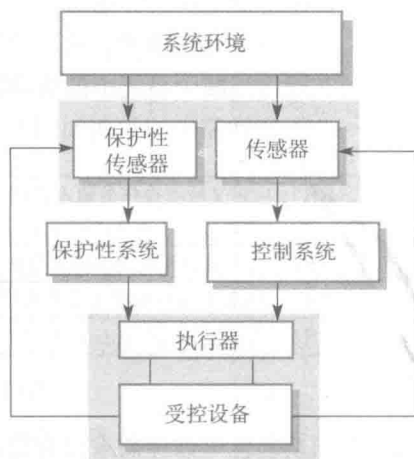


图 11-6 保护性系统体系结构

保护性系统只包含最紧要的功能,能够把系统从潜在的危险状态转换到安全状态(系统关闭)。它是一个更加通用的容错系统体系结构的实例,主系统由一个更小更简单的、只包含关键功能的备用系统所支持。比如,美国航天飞机控制软件有一个备用系统包含“带你回家”功能,即备用系统可以在主控制系统失效又没有其他控制功能时让飞船安全着陆。

这种体系结构类型的优势是,保护性系统的软件可以比控制被保护过程的软件简单得多。保护性系统的唯一功能是监视操作并确保紧急情况下系统被带回安全状态。因此,我们就可以投入更多的精力到容错和故障检测中去。你可以检查软件规格说明是否正确且一致,而且软件参照其规格说明也是正确的。目的就是确保保护性系统的可靠性,即在需要启动时其发生失效的可能是非常小的(如 0.001)。对保护性系统的调用应该是非常罕见的,如果请求失效率为 1/1000,意味着保护性系统失效应该是非常稀少的。

11.3.2 自监控系统体系结构

自监控系统体系结构(见图 11-7)是指这样一种系统体系结构:系统设计成监视其自身的操作,并在探测到问题的时候采取某些行动。这是通过在不同的通道计算并比较计算的输出来

实现的。如果输出是一致的并且同时可用，那么可以判定系统运行是正常的；如果输出不同，那么可以假设发生了失效。当其发生的时候，系统通常会在状态输出线上输出一个失效异常，这会使控制被转移到另一个系统上进行。

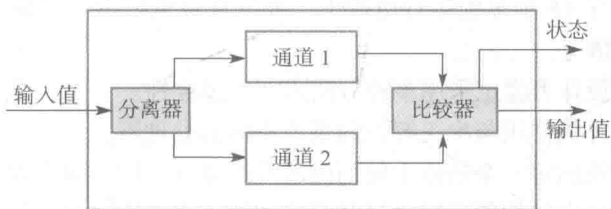


图 11-7 自监控系统体系结构

为了能有效地检查硬件和软件故障，自监控系统必须被设计成：

1. 每一个通道上使用的硬件都是不一样的。现实中，这可能意味着每一个通道使用不同的处理器类型来实现要求的计算，或者是组成系统的芯片集可能源自不同的生产商。这降低了通常的处理器设计故障影响计算的可能性。
2. 在不同的通道上使用不同的软件。否则相同的软件错误可能会在同一时间在每一个通道上发生。

对其本身而言，此体系结构可以用于如下情形：计算的正确性非常重要，但可用性并不是至关重要。如果从不同的通道获得的答案是不同的，系统会简单地关闭。对于很多医疗和诊断系统，可靠性要比可用性更加重要，因为错误的系统响应会导致病人接受错误的治疗。然而，如果系统在错误事件中仅仅是关闭了，这虽然会带来不便，但通常病人不会因此而受到系统的伤害。

在需要高可用性的情况下，需要并行使用几个自监控系统。需要一个切换单元来检查故障，并在有两个通道产生一致的结果时从一个系统取得结果。这样的方法被用在空中客车 340 系列飞行控制系统中，在这个系统中使用了 5 个自检查计算机。图 11-8 是空中客车飞行控制系统简化图，其中显示了自监控系统的组织结构。

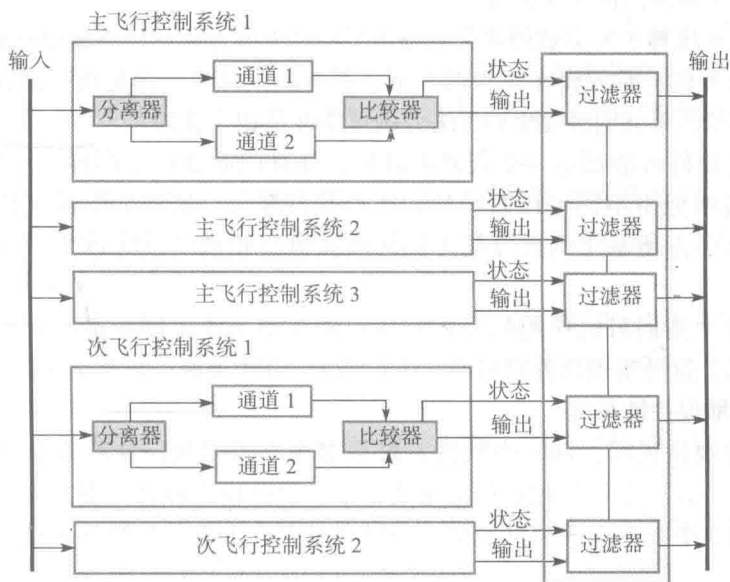


图 11-8 空中客车飞行控制系统体系结构

在空中客车飞行控制系统中, 每一个飞行控制计算机并行地计算, 使用的是同一个输入。输出被连接到一个硬件过滤器上以检查这个状态是否指示的是一个故障。如果是, 则这台计算机输出就被关闭。接下来从另一台替代计算机输出。因此, 4 台计算机都失效而飞行操作依然可能继续。在 15 年多的运行历程中, 并没有报道过由于所有的飞行控制系统失效而导致飞行器失控的情况。

空中客车系统的设计者尝试采用多种方法来获得多样性。

1. 主飞行控制计算机使用与次飞行控制系统不同的处理器。
2. 主系统和次系统的每一个通道中使用的芯片组来自不同的生产商。
3. 次飞行控制系统中的软件只提供最紧要的功能, 因此比主控制软件要简单。
4. 主系统和次系统的每一个通道的软件使用不同的程序语言来开发, 并且使用不同的开发团队。
5. 在主系统和次系统中使用不同的编程语言。

如 11.3.4 节要讨论的, 这些都不能保证多样性, 但是它们减少了不同通道共同失效的概率。

11.3.3 N 版本编程

自监控系统体系结构是采用多版本编程提供软件冗余和多样性的系统的例子。这个多版本编程的概念源自硬件系统中的三重模块冗余 (Triple Modular Redundancy, TMR) 的概念。这个概念已经在构建容许硬件失效的系统中用了很多年 (见图 11-9)。

在一个 TMR 系统中, 硬件单元被复制 3 次 (或者更多次)。每一个单元的输出被送到一个输出比较器上, 这个比较器通常实现为一个投票系统。这个系统比较所有的输入, 并且如果两个或者更多的输入是相同的, 那么其值就是输出。如果其中的一个单元失效了, 并且产生的输出与其他单元产生的输出不一样, 那么输出就被忽略。故障管理器可能自动尝试修改这个有故障的单元, 但如果做不到的话, 系统会自动地重新配置, 使那个发生故障的单元退出服务。系统会用剩余的两个单元继续工作下去。

这个容错方法依赖于大多数硬件失效是由于构件失效造成的而不是设计故障造成的。这些构件是独立失效的。它假设在正常情况下, 所有的硬件单元都是按规格说明那样去工作的。因此所有的硬件单元同时发生构件失效的可能性是相当低的。

当然, 这些构件可能都有一个普遍的故障, 并且因此都产生同样的 (错误) 答案。使用共同的规格说明但由不同生产商设计和生产硬件单元, 这样会降低这种共同模式失效发生的概率。这一点所基于的假设是不同的团队做出相同的设计或生产错误的概率是很小的。

在容错软件中我们可以使用相似的方法, 这就是 N 个不同版本的软件系统并行执行 (Avizienis 1995)。这种实现软件容错的方法在图 11-10 中说明, 其被用于铁路信号系统、航空系统以及反应堆保护性系统。

使用共同的规格说明, 同一个软件系统由多个小组实现, 这些版本在不同的计算机上执行, 使用一个投票系统比较它们的输出, 不一致的输出或者未能及时产生的输出被拒绝。至少系统的 3 个版本应该是可用的, 所以在发生单个失效的情况下两个版本应该是一致的。

在需要高可用性的系统中, N 版本编程可能比自监控体系结构要便宜一些。然而它仍然

需要多个不同的小组开发多个软件版本，这导致很高的软件开发费用。因此，仅在无法为安全关键系统构造保护性系统的时候，才使用这个方法。

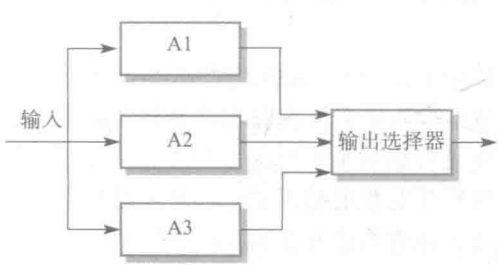


图 11-9 三重模块冗余

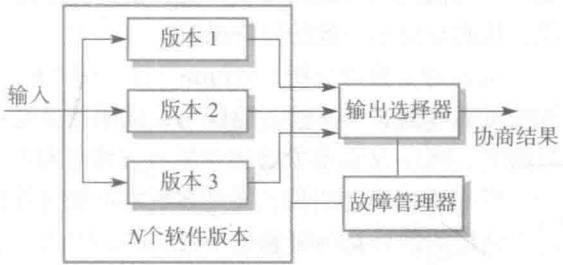


图 11-10 N 版本编程

11.3.4 软件多样性

所有以上的容错体系结构都依赖于软件的多样性以获得容错功能。这一点基于的假设是：同一规格说明（或者对于保护性系统来说是一部分规格说明）的不同实现之间是相互独立的。它们不应该包含同样的错误，这样也就不会以同样的方式同时失败。这需要软件由不同的小组来编写，而这些小组在开发过程中不应该交流，从而降低一样的误解或对规格说明误读的概率。

- 采购系统的企业可能包含明确的多样性政策，以期最大化系统版本之间的差异。比如：
- 1. 需求中明确要求使用不同的设计方法。比如，要求一个小组生产一个面向对象的设计，而要求另一个小组生产一个面向功能的设计。
 - 2. 规定程序要由不同的编程语言实现。比如，使用 Ada、C++ 和 Java 来写不同的版本。
 - 3. 要求对系统的开发要使用不同的工具和不同的开发环境。
 - 4. 明确要求在实现的某些部分用不同的算法。然而这限制了设计团队的自由，并可能因此难以与系统性能需求相吻合。

理想情况下，系统的多版本之间应该没有依赖性，所以应该以不同的方式失效。如果是这样，多样性的系统的总可靠性可以通过每一个通道的可靠性相乘获得。所以，如果每一个通道的请求失效概率是 0.001，则一个三通道系统（通道之间独立）总的 POFOD 比一个单通道系统的可靠性高 100 万倍。

然而现实当中，获得完全独立的通道是不可能的。实验证明独立的设计小组经常犯同样的错误或者对规格说明的同一部分都理解错了（Brilliant, Knight, and Leveson 1990; Leveson 1995）。这有以下几个原因：

- 1. 不同小组的成员经常来自同一个文化背景，并可能在受教育时使用同样的教科书。这意味着他们可能觉得很难理解的事情是类似的，并且可能同样觉得与领域专家交流很困难。很可能他们各自犯下的错误和设计的算法都是一样的。
- 2. 如果需求是不正确的，或者基于对系统环境的误解，那么这些错误将会反映到系统的每一个实现上。
- 3. 在关键系统中，详细系统规格说明是基于系统需求得到的，应该明确无歧义地描述系统行为。如果规格说明含糊不清，那么不同的团队可能以同样的方式曲解规格说明。

一种有可能减少共同规格说明错误发生的方法是开发详细的规格说明，即独立地开发详

细规格说明, 并使用不同语言进行描述。一个开发团队可能依据形式化规格说明来开发, 而另一个依据基于状态的系统模型来开发, 而第三个依据自然语言规格说明来开发。这可以避免一些规格说明理解的错误, 但无法避免需求错误的问题。在理解需求的时候也可能引入错误, 从而导致不一致的规格说明。

通过对实验的分析, Hatton (Hatton 1997) 总结得出: 一个三通道系统大约比单通道系统的可靠性高 5 ~ 9 倍。他认为, 试图通过对单个版本投入更多资源以增进可靠性是达不到目的的, 所以 N 版本方法比单版本方法更有可能产生更可靠的系统。

然而不能确定的是, 通过多版本增加可靠性的额外开发费用是否值得。对于很多系统来说, 这样的额外花销可能是不合理的, 因为一个设计良好的单通道系统或许已经足够了。只有在安全关键和任务关键的系统中, 由于失效的代价是高昂的, 才真正需要多版本软件。即使在这样的情况下 (比如航天系统), 提供一个有限的、功能简单的备份直到主系统能够被修复并且重启, 可能也就足够了。

11.4 可靠性编程

总的来说, 本书没有讨论编程问题, 因为在没有涉及编程语言的具体细节的情况下讨论编程问题几乎是不可能的。现在有很多方法和语言应用在软件开发过程中, 所以本书避免用某一种语言为例。然而, 当考虑到可靠性工程时, 有一套已经被接受的好的编程实践, 它们的应用相当普遍而且可以减少交付系统中的故障。

图 11-11 显示了一个好的实践指导准则的列表。它们可以应用在任何一种系统开发使用的语言中, 尽管它们所使用的方式取决于系统开发所使用的特定语言和符号系统。遵循这些准则还可以降低程序中出现安全相关的故障或漏洞的可能性。

准则 1: 限制程序中信息的可见性

军事机构中采用的一个信息安全原则是所谓的“需要知道”原则。只有那些需要了解某条信息以便执行任务的人才给予此条信息。若信息不与他们的工作直接相关就不会给予他们。

在程序设计过程中, 也要用类似的原则去控制访问系统所使用的变量和数据结构。程序构件应该只允许访问那些与自身实现相关的数据。其他程序数据应该是不可访问的, 对构件是隐藏的。如果使用了信息隐藏, 隐藏的信息就不会被无关构件所破坏。如果接口保持不变, 数据表示的改变将不会影响到系统中的其他构件。

在程序中把数据结构实现为抽象数据类型可达到上述目的。抽象数据类型的内部结构及其变量表示是隐藏的, 类型的结构和属性外部不可见, 对数据的所有访问要通过操作进行。

比如, 你可能用一个抽象数据类型来表示一个请求服务的队列。操作应该包括 `get` 和 `put`, 也就是向队列里加入成员项或者从队列取出成员项, 还要有获得队列中成员项数量的操作。可以在最初将这个队列实现为一个数组, 但接下来又将其改变为一个链表。这可以在不改变针对队列的任何代码的前提下实现, 因为队列的表示永远不会被直接访问。

可依赖性编程指南

1. 限制程序中信息的可见性。
2. 检查所有输入的有效性。
3. 为所有的异常提供处理程序。
4. 尽可能不使用容易出错的结构。
5. 提供重启功能。
6. 检查数组边界。
7. 调用外部构件时加入超时处理功能。
9. 为每一个表示现实世界值的常量命名。

图 11-11 可依赖性编程的良好实践指南

在一些面向对象的语言中,可以使用接口定义实现抽象数据类型,也就是声明一个对象的接口而不引用对象的实现。比如,可以定义一个接口 Queue,支持将对象加入队列、从队列取出对象、计算队列的大小等。在实现接口的对象类中,方法和属性都应该是类私有的。

准则 2: 检查所有输入的有效性

所有的程序都是从它们的环境中获取输入并处理它们。规格说明所做的假设是这些输入反映了它们在真实世界中的使用。比如,可以假设一个银行账户数字总是 8 位正整数。可是在很多情况下,系统规格说明没有定义在输入错误的情况下应该采取什么样的动作。人们不可避免地会犯错误,有时候会输入错误的数据。有时候,如在本书第 13 章将会讨论的那样,对系统的恶意攻击靠的就是错误的输入。当输入是从传感器或者其他系统来的时候,这些系统也可能出错并提供不正确的值。

因此,每一次从程序的运行环境中输入数据的时候都应该验证输入的有效性。很明显这些检查要取决于输入本身,但以下这些可能的检查还是会用到。

1. 范围检查。可以认为输入应该在一个特定的范围之内。比如,一个代表概率的输入应该在范围 0.0 ~ 1.0 之间,代表液态水温度的输入应该在 0 ~ 100℃ 之间,等等。

2. 位数检查。输入可以被看成是固定数量的字符的形式(比如 8 位字符表示的银行账户)。在其他情况,位数可能不确定但是可能有一个现实的最大上限。比如,人名的字符数应该不会超过 40 个字符。

3. 表示检查。输入应该是一个特定的样式,以一种标准的方式表示。比如,人名不应该包括数字,电子邮件由两部分组成并以 @ 符号分开,等等。

4. 合理性检查。输入是一系列输入中的一个,而你知道一些关于这些输入之间的关系,这样就可以检查输入值是否是合理的。比如,输入值代表一个家庭用电电表的读数,那么可以认为用电量应该大致与前一年同期用电量相仿。当然,变化是会有,但是巨大的变化往往意味着发生了问题。

输入有效性检查失败的时候,所要采取的措施取决于所实现系统的类型。在某些情况下,要将问题报告给用户并且要求重新输入这个值。如果一个值来自传感器,你可以使用最近的有效值。在嵌入式实时系统中,你可能必须根据历史信息来估计一个值,这样系统才能继续执行。

准则 3: 为所有的异常提供处理程序

在程序执行期间,错误或者难以意料的事件不可避免会发生。这些事情的发生或者是因为一个程序故障,或者是不可预料的外部环境的结果。在程序执行期间错误或不可预料的事件被称作“异常”。异常的例子可以是系统电源失效、企图访问不存在的数据,或者是数值上溢或下溢。

异常可能是由硬件也可能是由软件情况决定的。当一个异常发生的时候,必须由系统来处理。这一点可以由程序本身完成,或者在系统中包含一个对系统异常处理的转换控制机制。一般来说,系统的异常管理机制会报告错误并关闭执行。因此,为保证程序异常没有引起系统失效,应该为所有可能发生的异常定义一个异常处理程序,并保证所有的异常能被探测出来且得到明确的处理。

有一些编程语言,比如 Java、C++ 和 Python,包含支持异常处理的内部结构。当一种异常情况发生时,产生异常信号而语言运行时系统将控制转移到异常处理程序。这是一个声

明异常名字并对每个异常提供相应的处理动作的代码段（见图 11-12）。注意异常处理程序是在正常的控制流之外存在的，而这个正常的控制流不会在异常处理之后恢复执行。

异常处理程序通常做以下 3 件事情中的一件：

1. 向更高层构件发送信号报告一个异常已经发生，并提供异常类型的信息。当一个构件调用另一个构件的时候使用这个方法，发出信号的构件必须知道被调用的构件是不是成功执行了。如果没有，要由调用构件采取动作来从这个问题中恢复。

2. 实现一些代替处理来替换原始准备的处理。这样，异常处理程序要采取一些措施来从问题中恢复。处理过程能如平常一样继续，或者异常处理程序能指示一个异常已经发生以提醒调用构件问题的存在。

3. 将控制传递到能处理异常的编程语言运行时支持系统。当程序发生故障的时候（比如，发生一个数值溢出），这经常是默认的处理。通常运行时系统采取的动作是暂停处理过程。在将控制转给运行时系统之前，如果能将系统转移至一个安全和静止的状态，这是唯一应该使用的方法。

在程序中处理异常使得检测和恢复一些输入错误以及未预料到的外部事件成为可能。从某种意义上讲，它提供了一种容错层次——程序检测到故障并且能够采取动作进行恢复。因为大多数的输入错误和未预料的外部事件通常都是瞬态的，所以往往在异常得以处理之后可以继续正常运行。

准则 4：尽可能不使用容易出错的编程元素



容易出错的编程元素

与其他编程语言相比，一些编程语言特征更可能导致程序错误的引入。如果避免使用这些结构，程序可靠性可能会得到改善。只要有可能，应该尽量减少 go 语句、浮点数、指针、动态内存分配、并行、递归、中断、别名、无界数组和默认输入处理的使用。

<http://software-engineering-book.com/web/error-prone-constructs/>

程序中的故障源于人所犯的错误，它们进而造成很多程序失效。程序员因未能很好跟踪状态变量之间的很多关系而犯错误，他们写的程序语句导致未预料到的行为和系统状态改变。人们总是会犯错，但在 20 世纪 60 年代末期，人们逐渐认识到了有些编程方法比其他编程方法更容易向程序中引入错误。

例如，应该尝试避免使用浮点数，因为浮点数的精度会受到其硬件描述的限制。比较大或非常小的数是不可靠的。另一个可能出错的结构是动态分配内存，当该内存不需要时很容易忘记释放，这可能导致难以检测的运行时错误。

有些安全关键系统的开发标准完全禁止这些元素的使用。不过，这种极端做法有时也是不实际的。这些结构和技术是很有用的，但必须小心使用。只要有可能，就应该在抽象数据类型或对象中使用这类结构，从而控制潜在的危险。如果错误发生，抽象数据类型和对象就

代码段

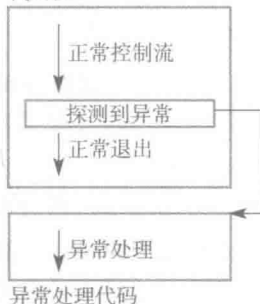


图 11-12 异常处理

担当了类似“防火墙”的角色,限制了损失的程度。

准则 5: 提供重启功能

很多机构信息系统都是基于短事务的,处理用户的输入需要较短的时间。这些系统被设计成仅当所有的过程都成功完成之后才做出对系统数据库的改动。如果在处理的过程中有错误,数据库将不会被更新,因此不会出现不一致的情况。事实上,对于所有的电子商务系统,只要是你在最后屏幕上确认你所购买的东西,都是以这种方式工作的。

用户与电子商务系统的交互总是持续几分钟时间并只包含很少的处理。数据库事务很短,并且通常可以在1秒内完成。然而其他种类的系统比如CAD系统和文字处理系统会包含长事务。在长事务系统中,开始使用系统到结束任务可能要花费几分钟甚至几小时。如果系统在长事务中发生失效,那么所有的工作可能都会丢失。相似地,在计算密集型系统中,比如一些科学计算系统,可能需要几分钟甚至几小时才能完成计算。如果发生系统失效,那么所有的时间都将白费。

在所有这类系统中,应该提供一个重启功能,目的是保持在处理过程中收集或生成的数据的副本。重启功能应该允许系统使用这些副本重新开始,而不是从头开始。这些副本有时被称为检查点。比如:

1. 在电子商务系统中,可以保存用户所填的表单,并允许他们访问和提交这些表单而无需让他们再次填写。

2. 在长事务或计算密集型系统中,可以每几分钟自动保存数据,并且在系统失效的时候使用最近保存的数据重新开始。同时,应该允许用户错误,提供一种用户能够返回到最近的检查点并从那里重新开始的方法。

如果异常发生而且正常的操作无法再继续下去,可以使用向后错误恢复来处理异常。这意味着要将系统状态重置为检查点处保存的状态,并从这个点重启操作。

准则 6: 检查数组边界

所有的程序语言都允许开发数组规格说明,即使用一个数字索引访问一个线性数据结构。这些数组通常设置在程序工作内存的一个连续区域。数组被描述成一种特定的长度,其长度反映出它们是如何被使用的。比如,要表示不超过10 000个人的年龄,那么可能会声明一个包含10 000个存储年龄数据的数组。

一些编程语言,比如Java,在每次对数组输入值的时候都做检查,检查索引是否在数组当中。所以如果一个数组A的索引是从0~10 000,试图访问元素A[-5]和A[12345]会导致异常的发生。然而编程语言如C和C++不会自动地进行数组边界检查,而仅仅简单地计算一个从数组的开始位置算起的偏移量。因此,A[12345]会访问从数组开始位置计算的第12345个位置的字符,不管这个字是不是数组的一部分。

这些语言不进行自动数组边界检查的原因是,这会在每次数组访问的时候引入额外开支。大部分数组访问是正确的,所以边界检查多数情况下是不必要的,并且会增加程序的执行时间。然而缺少边界检查会导致信息安全漏洞,比如第13章将介绍的缓冲溢出。更常见的是,它会向系统引入脆弱性而可能导致系统失效。如果你在使用没有数组边界检查的语言,如C或C++,你就要增加额外代码来保证数组的索引是在边界之内。

准则 7: 调用外部构件时加入超时处理功能

在分布式系统中,系统的构件在不同的计算机上执行,调用要在网络上从一个构件传送到另一个构件。为获得一些服务,构件A可以调用构件B。A在继续执行之前等待B的响

应。但是如果构件 B 由于某些原因没能做出响应,那么构件 A 就不能继续执行。它会为一个响应无限等待下去。等待系统响应的人看到的是一个沉默的没有任何响应的系统失效。没有别的办法,只能结束等待进程并重启系统。

为了避免这样,调用外部构件的时候应该总是包含超时机制。超时是自动地假设一次构件调用已经失效并不会产生响应。你要定义一个期待从被调用构件接收到响应的时间段。如果没有在这个时间段里接收到响应,可以认为调用已经失效并从被调用构件那里收回调用。你可以接下来尝试从失效中恢复,或告诉系统用户发生了什么事情和允许他们做什么。

准则 8: 为每一个代表现实世界值的常量命名

所有不是太简单的程序都包含一定数量的常量值来表示真实世界实体的值。这些值不会随着程序的执行而改变。有时候,它们是从不改变的常量(比如光的速度),但是更多的情况下,它们是随时间变化相对缓慢改变的值。比如,一个计算个人所得税的程序将会包含表示当前税率常量。这些常量将会年复一年地变化,所以程序必须用新的常量值进行更新。

应该在程序中加入一个片段,用来命名所有用到的现实世界常量值。当使用常量的时候,你应该用名字而不是值来指代它们。考虑到可依赖性,这样做有两点好处。

1. 可以少犯错误或少使用错误的值。我们很容易打错一个数字,系统往往不能发现这个错误。比如税率是 34%, 一个很简单的换位错误可能导致将其误打成 43%。但是如果你错打了一个名字(如标准税率),编译器通常都可以因未声明该变量而将其探测出来。
2. 当一个值改变的时候,你不用查找整个程序来发现在哪里使用过这个值。你所需要做的仅仅是改变与这个值相关的常量声明,新的值会自动在任何需要的地方出现。

11.5 可靠性度量

为评估系统可靠性,需要获取系统运行的数据。数据需求可能包括:

1. 对给定数量的系统服务请求统计系统失效的次数。这是用来测量 POFOD 指标的,并且不考虑做出请求的时间变化因素。
2. 系统失效间隔时间(或事务处理的数目),以及总的时间或者总的事务处理次数。这是用来测量 ROCOF 和 MTTF 的。
3. 系统失效导致不能提供服务之后的维修和重启所需的时间。这是用来测量可用性的。可用性不仅取决于失效间隔时间,而且取决于系统恢复运行的时间。

在这些度量中使用的时间单位可以是日历时间或诸如事务数量等离散单位。对于连续运行的系统,应该使用日历时间。监视系统,比如报警系统和其他种类的过程控制系统也归为这个类别。因此,ROCOF 可能是每天的失效数。处理诸如银行 ATM 或航空预订系统事务的系统的负载在一天之中会不断变化。在这些情况下,使用的“时间”单位可以是交易的数量;也就是说,ROCOF 将是每 N 千个事务中失败的事务的数量。

可靠性测试是度量系统可靠性的一个测试过程。已经有一些可靠性量度指标,像请求失效概率(POFOD)和失效发生率(ROCOF),用于量化所需软件的可靠性。如果系统已经达到了要求的可靠性水平,可以在可靠性测试过程中得到验证。

测量系统可靠性的过程有时被称为统计测试(见图 11-13)。统计测试过程适用于可靠性测量而不是故障查找。Prowell 等人(Prowell et al. 1999)在他们关于净室

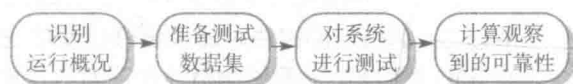


图 11-13 可靠性测量的统计测试

软件工程的书中给出了统计测试的良好描述。

在统计测试过程中有 4 个阶段。

1. 从研究已经存在的同一类型系统开始, 理解这些系统在实践中是怎样被使用的。这很重要, 因为必须通过系统用户的使用来度量其可靠性。目的是建立运行概况。运行概况要找出系统输入的分类, 以及这些输入将在正常使用中出现的可能性。

2. 构造一个能反映运行概况的测试数据集。这就是说要获得具有相同概率分布的测试数据作为所研究系统的测试数据。通常可以使用测试数据生成器得到测试数据。

3. 使用上面生成的测试数据对系统进行测试、记录发现的失效和每个失效类型发生的次数。正如第 10 章所讨论的, 所采用的时间单位应该适合相应的可靠性度量。

4. 当观察到相当数量的失效后, 软件的可靠性就可以计算了, 并且可以计算出适当的可靠性度量值。

这些可靠性测量方法虽然在理论上看起来很好, 但是在实际应用过程中并不是那么简单。

主要困难在于:

1. 运行概况的不确定性。运行概况可能无法精确反映系统真实的使用情况。

2. 测试数据生成的高成本。生成运行概况所需要的大量数据是非常昂贵的, 除非此过程完全能自动完成。

3. 高可靠性要求下的统计不确定性。所产生的故障和失效数量必须具有统计显著性, 从而确保可靠性测量的准确性。当软件已经足够可靠后, 会发生的故障相对较少并且很难产生新的故障。

4. 识别失效。系统失效的发生并不总是很明显。如果你有一个形式化的规格说明, 也许能够从中确定偏差。但是, 如果规格说明是自然语言, 由于其可能具有二义性, 观察者可能在系统是否失效方面存在分歧。

到目前为止, 产生大测试数据集的最好方法是使用某些测试数据生成器, 这些测试数据生成器能自动生成运行概况中规定的输入类型的测试数据。然而, 对于交互式系统却不总是能自动产生出所有的测试数据, 因为输入通常是对系统输出的响应。因而数据需要通过手工设计, 这项工作是费用很高的。即使有些地方可以实现完全自动化, 编写测试数据生成器的命令也要耗费大量时间。

统计测试可结合故障注入来收集数据, 以说明故障测试过程是如何有效执行的。故障注入 (Voas and McGraw 1997) 是有意地在程序中植入错误, 当程序执行时, 这些错误将导致程序故障并产生相关的失效。然后我们分析失效, 发现问题的根源是否是程序中所植入错误中的一个。如果发现 X% 的注入故障导致了失效, 那么故障注入的支持者会声称, 这表明故障测试过程也会在程序中发现 X% 的实际故障。



可靠性增长建模

可靠性增长模型是描述系统可靠性在测试过程中随时间变化的模型。当发现系统失效时, 导致这些失效的潜在故障被修复, 从而使系统的可靠性在系统测试和调试期间得到改善。为了预测可靠性, 概念化的可靠性增长模型必须转化为数学模型。

<http://software-engineering-book.com/web/reliability-growth-modeling/>

当然,这是假设注入故障的分布及类型和实际出现的故障是一致的。对由于编程错误产生的故障来说,这种假设应该是合理的,但是故障注入在预测源于需求和设计问题等其他原因导致的故障数量时并不有效。

11.5.1 运行概况

软件的运行概况反映软件在实际过程中将如何使用。它包含对输入类型的描述以及这些输入发生的可能性。当一个新的软件系统替换已存在的手工作业方式或自动化系统时,很自然地就能得到新软件可能的使用模式。新软件的使用模式应该大致上和现存的使用操作模式相仿,只是需要根据新软件所提供的功能进行适当调整。例如,对于一个电信交换系统,可以给出它的运行概况,因为通信公司知道这个系统所需要处理的调用模式。

运行概况中关于输入及其可能性的典型描述如图 11-14 所示。那些发生概率极高的少数输入类型列于图中的最左侧,还有大量输入类型,它们发生的可能性是相当低的,但又不是不可能发生的,这些输入类型列于图中的右侧,图中省略号表示该图并未详尽列出所有的输入类型。

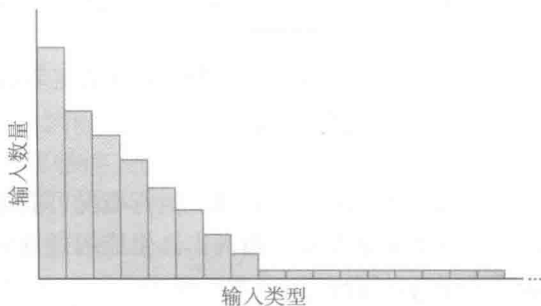


图 11-14 运行概况中的输入分配

Musa (Musa 1998) 讨论了他所从事的电信领域中运行概况的发展。因为收集用法数据已经有很长的历史了,所以运行概况的开发过程相对比较简单。文档仅仅反映了历史的用法数据。对于一个需要大约 15 人年方能开发完成的系统,运行概况大约需要 1 人月就可以完成。而在其他情形中,运行概况生成要长得多(2~3 人年),当然,这个成本最后分解到了系统多个发布版本中。

不过,当一个软件系统是新的或者有所创新的时候,预期它将如何使用是很困难的。因此,得到准确的运行概况是不太可能的。用户对系统有不同的期待,他们的背景不尽相同,而且他们也会有不同的经验,没有可以沿袭的用法数据库。计算机用户对系统的使用方式往往超出设计者的估计。

对于一些像电信系统这样使用标准化模式的系统来说,开发准确的运行概况是完全有可能的。然而,对于其他类型的系统,开发准确的运行概况可能是困难的或不可能的。

1. 系统可能有许多不同的用户,每个用户都有自己使用系统的方式。正如之前在本章的解释,因为用户以不同的方式使用系统,所以不同的用户有不同的可靠性印象。难以在单个运行概况中匹配所有的使用模式。

2. 用户会随时间的推移而改变他们使用系统的方式。随着用户学习一个新系统并对其更有信心,他们开始以更复杂的方式使用它。因此,在用户熟悉系统之后,与系统的初始使用模式匹配的运行概况可能不再有效。

由于这些原因,通常不可能开发一个完全可信赖的运行概况。如果你使用过时或不正确的运行概况,则你不能确信所做的任何可靠性测量的准确性。

要点

- 软件可靠性可以通过以下方法实现:避免引入故障,在系统部署之前检测和移除故

障，引入在系统出错后仍允许系统保持可操作的容错设施。

- 可靠性需求应该在系统需求规格说明中量化定义。可靠性量度包括请求失效概率 (POFOD)、失效发生率 (ROCOF) 以及可用性 (AVAIL)。
- 功能性可靠性需求是系统功能性的要求，例如检查和冗余需求，这有助于系统满足其非功能性可靠性需求。
- 可依赖系统体系结构是为容错设计的系统体系结构。有很多种体系结构风格支持容错，包括保护性系统、自监控系统体系结构和 N 版本编程。
- 实现软件多样性很难，因为现实中不能保证软件的每个版本都是真正独立的。
- 可依赖编程利用程序中包含冗余来检查输入的有效性和程序变量的值。
- 统计测试用来估计软件可靠性。它采用与运行概况相匹配的测试数据对系统进行测试，这个测试数据集反映软件在使用时的输入分布。

阅读推荐

《Software Fault Tolerance Techniques and Implementation》全面讨论了容错技术和容错体系结构。该书还覆盖了关于软件可依赖性的更一般性的问题。可靠性工程是一个成熟的领域，这里所讨论的技术仍然有现实意义。(L. L. Pullum, Artech House, 2001)

《Software Reliability Engineering: A Roadmap》这篇由软件可靠性方面的引领研究者所写的综述论文总结了软件可靠性工程的发展现状，并讨论了该领域的研究挑战。(M. R. Lyu, Proc. Future of Software Engineering, IEEE Computer Society, 2007) <http://dx.doi.org/10.1109/FOSE.2007.24>

《Mars Code》这篇论文讨论了好奇号火星探测器的软件开发中所使用的可靠性工程方法。其中依赖于良好的编程实践、冗余、模型检测（在第 12 章中介绍）的使用。(G. J. Holzmann, Comm. ACM., 57(2), 2014) <http://dx.doi.org/10.1145/2560217.2560218>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap11/>

支持视频的链接: <http://software-engineering-book.com/videos/reliability-and-safety/>

关于空中客车飞行控制系统的更多信息: <http://software-engineering-book.com/case-studies/airbus-340/>

练习

- 11.1 如果可靠性规格说明表示为系统整个生命周期中非常少量的失效，解释为什么验证这样的可靠性实际上是不可行的。
- 11.2 为下列软件系统提出合适的可靠性度量。为你的选择给出合适的理由。对这些系统的用法给出预测，并给出这些可靠性度量的合适的值：
 - 医院特护病房中监控病人的系统；
 - 字处理器；
 - 自动售卖机控制系统；
 - 汽车制动控制系统；
 - 制冷部件控制系统；

- 管理报告生成器。
- 11.3 如果火车在某个路段上的时速超过了限值，或者火车正在进入某个红灯亮的路段（例如，该路段不允许驶入）的时候，火车的保护系统就会自动工作。选择用于描述这个系统所要求的可靠性的可靠性度量，为你的答案给出理由。
- 11.4 什么是支持软件容错的体系结构风格的共有特性？
- 11.5 想象你在实现一个基于软件的控制系统。提出一种环境使其适合使用一个容错结构，解释为什么需要这种方法。
- 11.6 你负责设计一个通信交换机，必须提供 24/7 的可用性，但这不是一个安全关键系统。给出你答案的理由，并建议一种可以用在这个系统上的体系结构风格。
- 11.7 有人建议在治疗癌症的射线理疗仪的控制软件开发中使用 N 版本编程。你认为这是一个好的提议，说明原因。
- 11.8 为什么基于软件多样性设计的不同的系统版本都会经历同样的失败？
- 11.9 为什么应该在可用性要求极高的系统中对所有的异常进行明确处理？
- 11.10 软件失效可能对软件的用户造成相当大的不便。公司已知软件中包含可导致软件失效的故障，还发布这样的软件是否符合伦理？他们是否应该对用户因软件失效造成的损失负责？正如生产厂家需要对所卖的产品提供保修服务一样，法律是否应要求软件公司提供软件保修？

参考文献

- Avizienis, A. A. 1995. "A Methodology of N -Version Programming." In *Software Fault Tolerance*, edited by M. R. Lyu, 23–46. Chichester, UK: John Wiley & Sons.
- Brilliant, S. S., J. C. Knight, and N. G. Leveson. 1990. "Analysis of Faults in an N -Version Software Experiment." *IEEE Trans. On Software Engineering* 16 (2): 238–247. doi:10.1109/32.44387.
- Hatton, L. 1997. " N -Version Design Versus One Good Version." *IEEE Software* 14 (6): 71–76. doi:10.1109/52.636672.
- Leveson, N. G. 1995. *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.
- Musa, J. D. 1998. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill.
- Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.
- Pullum, L. 2001. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House.
- Randell, B. 2000. "Facing Up To Faults." *Computer J.* 45 (2): 95–106. doi:10.1093/comjnl/43.2.95.
- Torres-Pomales, W. 2000. "Software Fault Tolerance: A Tutorial." NASA. http://ntrs.nasa.gov/archive/nasa/casi./20000120144_2000175863.pdf
- Voas, J., and G. McGraw. 1997. *Software Fault Injection: Innoculating Programs Against Errors*. New York: John Wiley & Sons.

安全工程

目标

本章的目标是解释在开发关键系统时用来确保安全性的技术。阅读完本章后，你将：

- 理解安全关键系统的含义，以及在关键系统工程领域为何必须将安全性与可靠性分开考虑；
- 理解如何使用危险分析提取安全需求；
- 了解用于软件安全保证的过程和工具；
- 理解用于向监管机构证明系统安全的安全案例概念，以及在安全案例中如何使用形式论证。

在本书的 11.2 节中，简要描述了一架空中客车在华沙机场着陆时发生的空难。这起空难导致 2 人死亡，54 人受伤。随后的调查表明，这起事故的主要原因是控制软件的失效，降低了飞机制动系统的效率。这是一个罕见的、软件系统的行为导致死亡或受伤的例子。它说明，当今软件是许多系统的核心组成部分，对于维护和维持生命至关重要。这就是安全关键软件系统，人们已经为安全关键软件工程开发了一系列专门的方法和技术。

正如第 10 章中讨论的，安全性是主要的可依赖性属性之一。如果系统在没有灾难性失效的情况下运行，即没有导致或可能导致人员死亡或受伤的失效，则可以认为该系统是安全的。因为环境损害（例如化学品泄漏）可能导致后续的人身伤害或死亡，所以那些自身失效可能导致环境损害的系统也是安全关键的。

安全关键系统中的软件在实现安全性方面发挥着双重作用。

1. 因为系统一般是由软件控制的，使得软件做出的决定和随后的行动都是安全关键的。因此，软件行为与系统的整体安全性直接相关。

2. 软件广泛用于检查和监控系统中的其他安全关键构件。例如，所有飞机发动机部件由软件监控，以寻找部件失效的早期指示。如果当软件运行失效，其他构件可能会失效并导致事故的时候，这个软件就是安全关键的。

软件系统的安全性是通过了解可能导致安全相关失效的情况实现的。软件的设计使得这种失效不会发生。因此，你可能认为如果安全关键系统是可靠的并且按照指定的情况运行，则它将是安全的。不幸的是，并没这么简单。系统可靠性对于安全实现是必要的，但是还不够。可靠的系统可能是不安全的，反之亦然。华沙机场事故是这种情况的一个例子，本书将在 12.2 节中更详细地讨论。

可靠的软件系统可能不安全，有以下 4 个原因。

1. 我们不能百分百地确信软件系统是无故障的和能容错的。未被发现的故障可能潜伏很长一段时间，软件失效也许会发生在多年的可靠运行之后。

2. 需求规格说明可能是不完备的，因为它可能没有描述在一些关键时刻的必要的系统行为。很大一部分系统误操作来自于规格说明错误而非设计错误。在嵌入式系统错误的研究

中, Lutz (Lutz 1993) 总结说: 需求中的困难是引起安全类软件错误的主要原因, 它会一直存在直到集成和系统测试^①。Veras 等人 (Veras et al. 2010) 在空间系统领域最近的工作中证实“需求错误仍然是嵌入式系统的主要问题”。

3. 硬件故障可能会导致传感器和执行器行为发生异常。当构件与物理失效很紧密时, 它们会产生不规则的行为, 其产生的信号会超出软件所能处理的范围。之后, 软件就可能运行失败或错误地解释这些信号。

4. 系统的操作者给出的一些输入单独看起来可能是正确的, 但是在某些情形下它可能导致系统发生故障。一个有趣的例子是, 一架飞机在停机坪上起落架突然塌下。显然, 技术员按下了指示设施管理软件收起起落架的按钮。软件完美地执行了技术员的指令。然而, 系统应该不允许该指令的执行, 除非飞机已经升空。

因此, 在开发安全关键系统时, 必须同时考虑可靠性和安全性。第 11 章中介绍的可靠性工程技术显然适用于安全关键系统工程。因此, 这里不讨论系统架构和可依赖的编程, 而是侧重于能改进和确保系统安全的技术。

12.1 安全关键系统

对于安全关键系统而言, 系统总是安全运转是至关重要的。也就是说, 系统永远也不能伤害人或者损害系统环境, 而无论系统是否符合其规格说明。安全关键系统的例子包括飞机的监控系统、化学和医学工厂中的过程控制系统, 以及汽车控制系统。

安全关键软件分为两大类。

1. 主要的安全关键软件。这种软件是嵌入在系统控制器中的软件。该类软件的错误执行会导致硬件的误操作, 引起人员伤亡或环境破坏。如在第 1 章中介绍的胰岛素泵软件, 就是主要安全关键系统。系统失效会导致用户受到伤害。

胰岛素泵系统是一个简单的软件控制系统, 但是其他非常复杂的安全关键系统也依赖软件来控制。软件控制比硬件控制更重要, 因为大量的传感器和执行机构被软件管理, 这些单元都有复杂的控制准则。例如, 先进的空气动力军用飞机失去平衡的时候, 要求连续对飞行姿态做出软件控制的调整, 以确保不会发生坠毁事件。

2. 次要安全关键软件。此类软件可以间接引起人员伤亡。此类软件的典型例子是计算机辅助工程设计系统, 它的错误执行会引起所设计的对象中存在设计故障。如果所设计的系统误操作的话, 这样的故障会引起对人的伤害。另外一个例子是心理治疗管理系统。由于该系统失效, 一个病情不稳定的病人可能没有得到正确的治疗, 使得该病人伤害了自己或他人。

一些控制系统, 例如控制关键国家基础设施 (电力供应、电信、污水处理等) 的控制系统是次要安全关键系统。这些系统的失效一般不会立即对人类产生严重后果。然而, 受控系统的长时间中断可能导致受伤和死亡。例如, 污水处理系统失效可能导致更危险的传染病, 因为未处理的污水被排放到环境中。

第 11 章中解释了如何通过故障避免、故障检测和排除、容错来实现软件和系统的可用性和可靠性。安全关键系统的开发在使用这些方法的同时, 通过考虑可能发生的潜在系统事故, 利用危险驱动技术来增强这些方法。

1. 危险避免。系统的设计要能避免危险的发生。例如, 切纸系统要求操作员使用双手同

① Lutz, R R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In RE'93, 126-133. San Diego CA: IEEE. doi:0.1109/ISRE.1993.324825.

时分别按住一个按钮，以避免操作员的手在切刀的路径上。

2. 危险检测和排除。系统的设计要能检测危险，并在其发生之前去除危险。例如，一个化学工厂系统会检测过大的压力并开启减压阀在爆炸发生之前降低这些压力。

3. 限制损失。系统中可能会包含一些保护性的特征，从而尽量减少事故带来的损失。例如，正常情况下飞机引擎带有自动的灭火器。如果失火，通常可以在其对飞机造成威胁之前得到控制。

危险是可能导致事故的系统状态。使用上述切纸系统的示例，当操作者的手处于切割刀片能够伤害到的位置时，危险出现。危险不是事故，我们经常让自己陷入危险的情况，但是总能毫发无损地脱离危险。然而，发生事故之前总是存在危险，因此减少危险可减少事故。

危险是安全关键系统工程中使用的专业词汇的一个例子。图 12-1 中解释了安全关键系统中使用的其他术语。

术 语	定 义
事故（意外）	未预计到的事件和事件后果，它导致人员伤亡、财产损失或环境破坏。过量注射胰岛素是事故的例子
损失	关于意外所造成结果的一个度量。损失可以是很多人在事故中遇难，也可能是很小的身体外伤或很小的财产损失。胰岛素过量可能导致很严重的损害（损失）甚至引起患者的死亡
危险	潜在能引起或造成事故的情况。测量血液血糖含量的传感器失效是危险的一个例子
危险概率	发生危险的可能性。概率值往往是任意的，但是可以用“可能”（有 1/100 的危险发生概率）到“不太可能”（没有可想到的发生危险的情形）等来衡量危险可能。在胰岛素泵中传感器失效导致过量注射的概率就是“较低”
危险严重性	因特定的危险所导致的最坏可能损失的评估。危险严重程度可以是灾难性的，很多人遇难；也可能是很小的，如只有微不足道的损失。只要有可能造成人的死亡，那危险程度就将是“非常高”
风险	这是系统会发生事故的程度的度量。风险评估考虑了危险概率、危险严重性，以及危险演化为事故的概率。胰岛素过量的风险是“中度”到“低”

图 12-1 安全性术语

事实上，我们现在很擅长构建能够应对单个问题的系统。我们可以在系统中设计能够检测和恢复单个问题的机制。然而，当几件事同时出错时，事故更可能发生。随着系统变得越来越复杂，我们无法理解系统的不同部分之间的关系。因此，无法预测许多意外的系统事件或失效组合所导致的后果。

关于严重事故的分析，Perrow（Perrow 1984）表明，它们几乎都是源于系统不同部分失效的组合情形。子系统失效的未预料的组合导致整个系统交互的失效。例如，空调系统的失效可能导致过热，而这又会导致系统硬件产生不正确的信号。

Perrow 指出，在复杂的系统中，不可能预测所有可能的失效组合。因此，他提出了“正常事故”这一短语，意味着当我们建立复杂的安全关键系统时，事故必须被认为是不可避免的。

为了降低复杂性，我们可以使用简单的硬件控制器而不是软件控制。然而，相对于机电系统，软件控制系统会监视更大范围的状态。它们也更容易进行修改，使用的计算机硬件具有非常高的内在可靠性，也在物理上减少了体积和重量。

软件控制系统能提供复杂的安全互锁机制，可以支持相应策略以减少人们在危险环境中停留的时间。尽管软件控制会导入更多的错误，但它也允许更好的监控和保护，因而对改善

系统安全性有帮助。

很重要的一点是保持对于安全关键系统的感知。关键软件系统大多数时候运行都是没有问题的。整个世界上只有很少量的人由于软件故障而死亡或受伤。系统不可能是百分百安全的，我们必须确定是否值得为使用先进技术所带来的好处而忍受意外事故所带来的风险。



基于风险的需求规格说明

基于风险的需求规格说明是安全性和信息安全性要求极高的系统的开发者广泛使用的方法。它注重那些可能造成最大损失或可能经常发生的事件。仅造成不严重的后果或者极少发生的事件可能被忽略。基于风险的需求规格说明过程包括：理解系统所面对的风险，发现它们的根源，生成需求来管理这些风险。

<http://software-engineering-book.com/web/risk-based-specification/>

12.2 安全需求

在本章的引言中描述了华沙机场的飞行事故，事故里空客的制动系统发生故障。对于事故的调查显示制动系统软件按其规格说明正常工作了。程序中是没有错误的。然而，软件的规格说明是不完备的，它没有考虑到极特殊的情况，恰巧这种极特殊情况发生了。软件正常工作但是系统失败了。

这个事件说明了系统的安全性不仅依赖于好的工程过程，还需要在制定系统需求时关注一些细节，以及用来保证系统安全性的特殊的软件需求。安全需求是功能性需求，功能性需求定义了系统应该包含的检查和修复措施，以及防止系统失效和外部攻击的保护性特征。

产生功能性安全需求的起始点通常是领域知识、安全标准或法规。它们是高级别的需求，最好用“不应该”需求来描述。与定义系统应该做什么的功能性需求相比，“不应该”需求定义系统不能被接受的行为。“不应该”需求的例子是：

“当飞机在飞行时，系统不应该允许选择反向推力模式。”

“系统不应该允许3个以上报警信号同时被激活。”

“导航系统不应该允许用户在汽车移动时设置所需的目的地。”

这些“不应该”需求不能直接实现，而是必须分解成更多的比较专门的软件功能性需求。它们可能通过系统设计决策来实现，这样的设计决策的例子是，决定在一个系统中使用某种装置。

安全需求首要的是保护性需求，与普通的系统操作没有关系。它们可能明确说明系统应该被关闭，这样安全性才能得以保证。在所得到的安全需求中，需要在安全性和功能性之间找到一个可接受的平衡来避免过度保护。在花费合理的前提下研究构建非常安全的系统才是有意义的。

基于风险的需求规格说明是在关键系统工程中使用的一种通用方法，该方法识别系统面临的风险，并避免或减轻这些已识别的风险。它适用于所有类型的可依赖性需求。对于安全

关键系统，它转换为由识别的危险驱动的过程。正如上一节中讨论的，危险是指可能（但不是必需）造成人员伤亡的情形。

在危险驱动的安全规格说明过程中有4个活动。

1. 危险识别。危险识别过程识别可能威胁系统的危险。这些危险可以记录在危险日志中。这是一份记录安全分析和评估的正式文件，可作为安全案例的一部分提交给监管机构。

2. 危险评估。危险评估过程决定哪些危险是最严重的或最可能发生的。在确定安全需求时应优先考虑这些因素。

3. 危险分析。这是一个根本原因分析的过程，用于识别可能导致危险发生的事件。

4. 风险降低。这个过程基于危险分析的结果并识别安全需求。这些需求可能是关于确保一个危险不会引起或导致事故，或如果一个事故发生了将相关的损失最小化。

图12-2说明了这种危险驱动的安全需求规格说明过程。

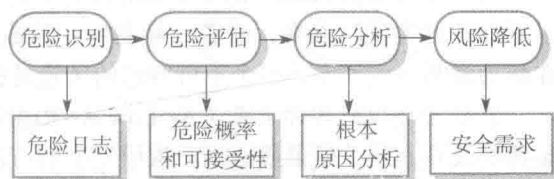


图12-2 危险驱动的需求规格说明

12.2.1 危险识别

在安全关键系统中，危险识别始于确定不同的危险类别，如物理危险、电子危险、生物危险、辐射危险、服务失效危险等。接下来分析每一类危险以发现可能发生的特定的危险。同时还必须识别由于不同危险可能的组合而产生的潜在危险。

有经验的工程师、领域专家以及专业的安全顾问一起工作，从先前的经验和应用领域中识别出系统危险。集体工作方式比如头脑风暴，对识别危险有很好的作用。对先前事故有直接经验的专门分析人员也能够识别特定的危险。

第1章中介绍的胰岛素泵系统的例子就是安全关键系统，因为失效可能导致患者受伤甚至死亡。当使用机器时可能发生的事故包括：用户承受长时间的低血糖的后果（眼睛、心脏和肾脏问题）；由低血糖引起的感知功能障碍；一些其他医疗情况，比如过敏反应。

胰岛素泵系统引起的一些危险包括：

- 胰岛素药量过大（服务失效危险）；
- 胰岛素药量不够（服务失效危险）；
- 硬件监控系统失效（服务失效危险）；
- 电池用尽造成停电（电器危险）；
- 机器与其他医疗设备发生电子干扰，如与心脏起搏器相互影响（电器危险）；
- 由于不正确的安装造成传感器和执行机构接触不良（物理危险）；
- 机器的某个部分在病人身体内脱离（物理危险）；
- 由于引入机器造成感染（生物危险）；
- 患者对机器材料或胰岛素的过敏反应（生物危险）。

软件相关的危险一般涉及系统服务失效，或者是监控系统或保护系统失效。监控系统和保护系统会检测出潜在的危险状态，比如断电。

可以使用危险日志（hazard register）记录所识别的危险，并解释为何该危险会被囊括其中。危险日志是一个重要的法律文件，它记录与每个危险相关的所有安全决策。它可以用

来证明需求工程师在考虑所有可预见的危险时已经十分注意和小心，并且已经分析了这些危险。在发生事故时，危险日志可用于随后的查询或法律诉讼中，以表明系统开发人员在系统安全分析中没有疏忽。

12.2.2 危险评估

危险评估过程注重理解危险发生的频率和危险发生所造成的后果。你需要进行分析来理解一个危险对系统或环境而言是不是严重威胁。分析同样还要提供基本信息以决定如何管理与危险相关的风险。

对于危险，分析和分类过程的结果是可接受性报告。报告是用风险术语表述的。风险包含意外发生的可能性和它的后果。在危险评估中可以使用3种风险类型。

1. 不可容忍的风险。在安全关键系统中是指那些能威胁到人的生命的危险。系统设计要求不能让这类危险发生，或者一旦发生，系统特征将保证在引起事故之前就已探知到它们。在胰岛素泵的例子中，一个不可容忍的风险是胰岛素过量注射。

2. 低于合理的实际水平（As Low As Reasonably Practical, ALARP）的风险。此类风险是那些没有太严重后果或后果严重但发生的可能性非常低的危险。系统设计应该让危险引起的事故尽量少，并满足成本或交付等因素的约束。胰岛素泵的一个低于合理的实际水平的风险可能是硬件监控系统的失效。其后果是，在最坏的情况下，短期的胰岛素量过低。这是一个不会引起严重后果的情况。

3. 可接受的风险。此类风险是指那些相关联的事故通常引起很小的损失。系统设计者应该采取所有可能的步骤来尽量降低可接受的风险，只要不会提高成本、延长交付时间或影响其他非功能性的系统属性。在胰岛素泵的例子中，一个可接受风险可能是引起用户一次过敏反应。这通常仅仅引起很小的皮肤发炎。它不值得为了降低这个风险使用特别的、更昂贵的材料与设备。

图12-3显示了这3个区域。三角形的宽度反映了确保风险不会导致事故的成本。最高的成本对应图顶部的危险，最低的成本对应三角形顶点处的危险。

图12-3中区域之间的边界不是固定的，而是取决于将部署系统的国家中风险的可接受程度。不同国家会有所不同——一些国家比其他国家更厌恶风险或更易引发诉讼。然而，随着时间的推移，所有国家都变得越来越厌恶风险，所以边界向下移动。对于罕见事件，接受风险的成本和为任何由此产生的事故支付的财务成本可能低于事故预防的成本。然而，公众舆论可能要求用钱来减少系统事故的可能性，而不考虑成本。

举例来说，对于一个公司，对偶然发生的污染事故的处理成本要比安装防止污染的装置的成本小。然而，公众和媒体不会容忍这样的事故，不预防而选择清理污染就变得不再能够接受。这样的事件可能还会导致风险的重新分类。比如，认为不太可能发生的风险（因此在

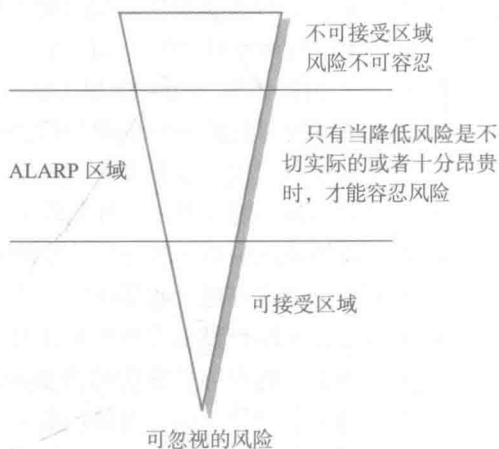


图 12-3 风险三角形

ALARP 区域中)可能因外部事件发生而重新划分为不可容忍,比如恐怖袭击,或自然现象,如海啸。

图 12-4 给出一个风险分类,对应前面章节中讨论的胰岛素注射系统中识别出的危险。把由于计算错误导致的胰岛素过量与胰岛素量过低的危险区分开来。一次胰岛素过量在短期内潜在地存在比胰岛素量过低更高的危险。胰岛素过量可以导致感知功能障碍、昏迷甚至死亡。胰岛素量过低引起血糖过高,在短期内会导致疲劳但是不会非常严重;然而长期来看,可能导致严重的心脏、肾脏以及眼睛问题。

识别出的危险	危险概率	事故严重性	估计的风险	可接受性
1. 所计算的胰岛素过量	中度	高	高	不可接受
2. 所计算的胰岛素不足量	中度	低	低	可接受
3. 硬件监控系统失效	中度	中度	低	ALARP
4. 电源失效	高	低	低	可接受
5. 机器未正常安装	高	高	高	不可接受
6. 机器失灵	低	高	中度	ALARP
7. 机器引起感染	中度	中度	中度	ALARP
8. 电子干扰	低	高	中度	ALARP
9. 过敏反应	低	低	低	可接受

图 12-4 胰岛素泵系统中的风险分类

图 12-4 中的危险 4 ~ 9 与软件无关,但是软件仍然要承担危险探测的角色。监控硬件的软件应该监控系统状态并警告潜在的问题。通常,警告在危险导致事故之前被检测出来。可以被检测出来的危险的例子有:通过监控设备电池检测到的电池失效,通过监控来自血糖传感器的信号检测到的血糖传感器放置不正确。

当然,这个监控软件是与安全相关的,因为危险检测失效会导致事故发生。如果监控系统失效但是硬件正常工作,那么这就不是一个严重失效。然而如果监控系统失效以至于硬件失效没有被检测到,那么就会导致更加严重的后果。

危险评估过程包括估计危险概率和风险的严重性。由于危险和事故不常见,所以危险评估过程通常是困难的,很多工程师没有危险事件或事故的直接经验。概率和严重性使用一些相对性的用词,例如“很可能”“不太可能”“罕见的”和“高”“中”“低”等。因为没有足够的危险事件和事故作为统计分析数据,所以不可能量化这些术语。

12.2.3 危险分析

危险分析是指发现安全关键系统中危险根源的过程,目标是找出什么事件或事件的组合可能会导致引发危险的系统失效。为做到这一点,可以使用自顶向下或者自底向上的研究方法。演绎性的自顶向下的技术可能会更容易使用一些,这种技术从危险开始分析到可能的系统失效。归纳性的自底向上的技术从一个所提出的系统失效出发,识别该失效可能会导致什么样的危险。

人们提出了各种各样的危险分解与分析技术 (Storey 1996)。最常用的技术之一是故障树分析,这是一种自顶向下的技术,用于分析硬件和软件危险 (Leveson, Cha, and Shimeall 1991)。这种技术很易于理解,并不需要专业领域知识。

要做故障树分析，先要从所有已经识别出的危险开始。对于每一个危险，要通过回溯发现可能引起该危险的原因。将危险置于树的根节点，并识别出所有能导致此危险的系统状态。接下来对每一个这样的状态，找出更多能导致该状态的系统状态。继续分解直到达到风险的所有根本原因。较之那些只有一个根本原因的危险，由一组根本原因的组组合引起的危险导致事故发生的可能性要小得多。

图 12-5 是关于胰岛素泵系统中的软件相关危险的一棵故障树。这些危险能导致不正确的胰岛素传输剂量。在这种情况下，把胰岛素不足与胰岛素过量合并为一种危险，即“不正确的胰岛素剂量”。这减少了所需要的故障树的数量。当然，在指定软件应该如何对这些危险做出反应的时候，必须区分胰岛素低剂量与胰岛素过量的区别。如前面所讲，它们不是同等重要——短期来看，过量有更大的危险。

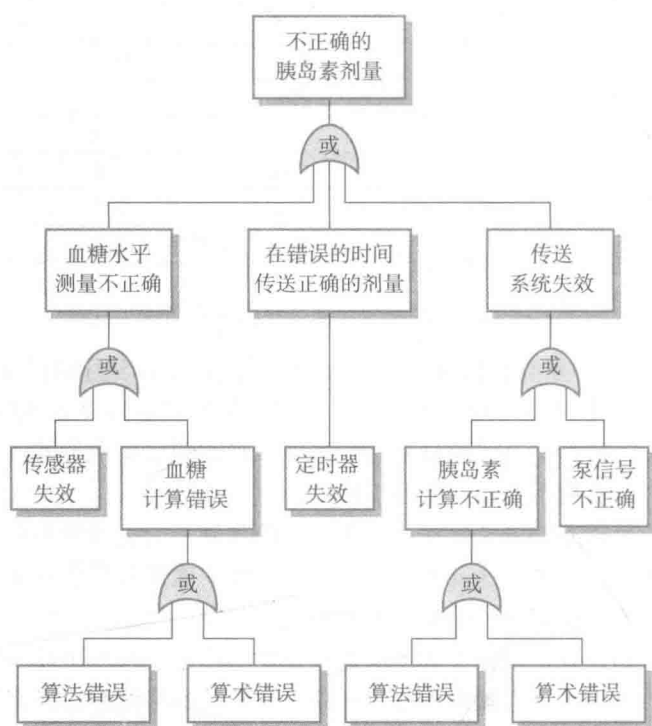


图 12-5 故障树的例子

从图 12-5 可以看到：

1. 有 3 种情况可以导致不正确的胰岛素剂量使用。血糖浓度可能被错误测量，所以胰岛素需求量也因错误的输入而被错误计算。传送系统可能对指定胰岛素注射量的指令没有正确反应。另外，剂量可能计算正确但是被过早或过迟传送。

2. 故障树的左枝是血糖值水平的错误测量。这可能是因为提供一个输入以计算血糖水平的传感器失效了，也可能是因为对血糖水平的计算错误。血糖值的计算是根据多个测量得到的参数，比如皮肤的传导性。不正确的计算可能是由错误的算法或者一个算术错误引起的，此错误可能是浮点数引起的。

3. 该树的中间分枝是关于时间性问题的，这些问题只能由系统定时器失效引起。

4. 故障树的右枝是关于传送系统失效的，检查这个失效的可能原因。这些原因可能来自

对胰岛素需求的不正确的计算，或者是因为没有能够传给胰岛素泵正确的信号。同样，一次不正确的计算可能源于算法错误或算术错误。

故障树也用在对潜在硬件问题的识别上。硬件故障树还可帮助理解软件需求，以检测和修正问题。例如，胰岛素的提供频率不会是很高的，一般不超过每小时3次，有时会更少。因此处理器有能力运行诊断和自检程序。在对患者产生严重后果之前，硬件错误如传感器、泵或者定时器错误就可以被发现并报警。

12.2.4 风险降低

一旦识别出潜在的风险及其根源，我们就能够得出安全需求以管理风险，并确保不会发生各种事故。可以使用3个可行的策略：

- 1. 危险避免。设计系统使得危险不能发生。
- 2. 危险检测和排除。设计系统使得可以在危险导致事故之前检测出危险并排除它。
- 3. 灾害限制。设计系统使得事故后果影响达到最小。

通常，安全关键系统的设计者会综合使用这些方法。一般来说，对于不可容忍的危险，要最大可能减少其发生的可能性，并增加一个保护系统以防这个危险发生（见第11章）。例如，在医药工厂控制系统中，系统将努力检测和避免反应器压力过高。不仅如此，还会提供一个独立的保护系统来监视压力，并当检测到压力过高的时候自动开启减压阀。

在胰岛素注射系统中，“安全状态”是没有胰岛素注射的关闭状态。在短时间内这不会对病人健康带来威胁。对于可能导致胰岛素泵系统运行错误的软件失效，需要下面这些“方案”。

- 1. 算术错误。该错误发生在一次算术计算引起一个表示失效，在系统规格说明中一定要识别出所有可能的算术错误，并为每个可能发生的错误设置异常处理程序。系统规格说明还要给出每当这种错误发生时所应该采取的行动。默认的安全行动会停止传送系统并启动一个警告。
- 2. 算法错误。这是一种更困难的情形，因为没有明显的必须处理的程序异常。通过比较要求的胰岛素剂量和之前传输的剂量，能检测到这类错误。如果高出很多，可能表明计算出的剂量是错误的。系统也可以继续跟踪剂量序列，在有多个高于平均值的剂量被传送时，就发一个警告并限制进一步的传送。

针对胰岛素泵系统，我们所得到的部分安全需求列于图12-6中。它们是用户需求，因此，应该更详细地表述在最后的系统规格说明中。

SR1：系统所传送的胰岛素单个剂量不应该大于一个指定的最大值。
SR2：系统所传送的胰岛素每日总剂量不应该大于一个指定的最大值。
SR3：系统应该包括一个硬件诊断工具，该工具在每小时内应执行不少于4次。
SR4：系统应该包含异常处理程序，异常处理程序处理系统中所有识别出的异常，参见表3。
SR5：当任何硬件或软件异常发生时，报警器就会发出响声，同时诊断消息（如表4中所列）也会显示出来。
SR6：在报警事件中，胰岛素传送应暂停，直到用户重新设定系统并清除报警。

注：表3和表4相关的表格都在需求文档中，此处并不列出。

图 12-6 安全需求的例子

12.3 安全工程过程

开发安全关键软件的流程基于软件可靠性工程。总的来说，要把中心放在开发一个完整

而且非常详细的系统规格说明上。系统的设计和实现常常遵守基于计划的瀑布模型,并且在每一个阶段都进行评审与检查。故障避免与故障检测是每个阶段的驱动。对于一些典型的系统,例如在第11章所讨论的航空系统,需使用容错体系结构。

可靠性是安全关键系统的先决条件。由于系统失效将会带来高成本与潜在危害,因此安全关键系统开发工作常常增加额外的验证活动。这些活动包括开发正式软件模型,分析并找到错误和不一致,以及使用静态分析软件工具解析软件源码以发现潜在错误。

安全系统必须可靠,但是,仅仅具有可靠性还不够。需求和验证如果有错误或者有遗漏,则可能意味着可靠系统是不安全的,因此,安全关键系统开发流程应包括安全评审,即由工程师和系统利益相关者检查已经完成的工作,尤其注意可能会影响系统安全的潜在问题。

正如在第10章中所讨论的,一些类型的安全关键系统是要遵守某些规定的。国家和国际监管机构需要详细的证据表明该系统是安全的。这些证据可能包括:

1. 已开发的系统规格说明和针对该规格说明所做的检查记录。
2. 已实施的验证和确认过程的证据,以及系统验证和确认的结果。
3. 那些表明开发该系统的组织有已定义和可依赖的软件过程的证据,还包括安全保证评审。也必须有记录显示这些过程已被适当地制定。

不是所有安全关键系统都是被监管的。例如,尽管现在汽车有很多嵌入式计算机系统,但汽车并没有监管机构。汽车系统的安全是汽车制造商的责任。然而,因为在发生事故时很可能会采取法律行动,非监管系统的开发人员必须保持相同的详细安全信息。如果对他们提起诉讼,他们必须证明在开发汽车软件时没有疏忽大意。

以上这些都要求大量的过程和产品文档,这也是敏捷过程如果不针对安全关键系统开发进行大幅的改动就无法在这种系统开发中使用的另一个原因。敏捷过程关注软件自身并且(合理地)认为大量的过程文档在产生之后并未实际使用过。然而,当不得不出于法律和监管的原因保留相关记录时,你就必须保持对于所使用的过程以及系统自身的文档记录。

安全关键系统,像其他具有高可依赖性需求的系统一样,需要基于可靠的过程(见第10章)。一个可依赖的过程通常都会包含以下活动:需求管理、变更管理和配置控制、系统建模、评审以及审查、测试计划、测试覆盖分析。对于一个安全关键的系统,还需要额外的安全保证、软件验证以及系统分析过程。

12.3.1 安全保证过程

安全保证是检查系统是否安全运行的一组活动。专门的安全保证活动一定要包含在软件开发过程的所有阶段当中。这些安全保证活动记录所进行的分析以及负责分析的人员。安全保证活动一定要全程用文档记录。这些文档用来向监管人员和系统所有者证明该系统能安全运行。

下面是一些安全保证活动的实例。

1. 危险日志和监控,从初步危险分析到测试再到系统确认,全过程跟踪危险。
2. 安全评审,贯穿于整个开发过程中。
3. 安全认证,对安全关键构件进行正式的安全认证。由独立于系统开发团队的外部人员组成小组,检查相关的证据,并且在投入使用前决定一个系统或构件是否是安全的。

为了支持这些安全保证过程,应该指定项目安全工程师,他们应当对系统的安全性负有明确的责任。这就意味着这些人在系统安全相关失效出现时应该承担责任。他们必须能够证

明安全保证相关步骤已经得到了正确的执行。

安全工程师和质量经理一起工作，确保使用一个详细的配置管理系统追踪所有安全相关文档，并且使之与相关的技术文档同步。如果配置管理失效意味着将错误的系统交付给了客户，那么严格的确认过程也就没什么意义了。质量和配置管理将分别在第 24 章和第 25 章中讲述。

危险分析是安全关键系统开发过程中的一个重要部分。危险分析的重点是识别危险、危险发生概率以及一个危险导致发生事故的的概率。如果有程序代码检查并处理每个危险，就可以证明这些危险不会造成事故。在一个系统使用前需要得到外部认证的时候（例如，飞机中的软件系统），能证明软件具有可跟踪能力通常是认证的前提条件。

应该生成的核心安全文档是危险日志。这个文档提供了有关如何识别在软件开发过程中所考虑的危险的证据。这个危险日志要用于软件开发过程的每个阶段，记录在各个开发阶段是如何考虑这些危险的。

胰岛素传送系统的一个简化的危险日志条目如图 12-7 所示。它记录了危险分析的过程，并显示了在这个过程中生成的设计需求。该设计需求的目的是确保控制系统不会向患者传送过量的胰岛素。

每个安全负责人都要在条目中明确地标明，这是很重要的，因为有以下两方面的原因。

- 1. 当人员被确定时，他们可以对自己的行动负责。这就意味着他们很可能更关心自己的工作，因为任何问题都可以追踪到他们的工作上。
- 2. 在发生事故时，很可能产生法律诉讼或调查。能够确认对安全保证负责的人很重要，这样他们就能够说明他们的行为。

危险日志		第 4 页：打印日期 2012-02-20			
系统：胰岛素泵系统		文件：InsulinPump/Safety/HazardLog			
安全工程师：James Brown		日志版本：1/3			
识别出的危险	向患者传送了过量的胰岛素				
识别人	Jane Williams				
危险类型	1				
识别出的风险	高				
故障树识别	是	日期	2011-01-24	位置	危险日志第 5 页
故障树创建人	Jane Williams 和 BillSmith				
故障树检查	是	日期	2011-01-28	检查人	James Brown
系统安全设计需求					
1. 系统必须包含自检测软件，能够测试传感器系统、时钟以及胰岛素传送系统。					
2. 自检测软件必须每分钟执行一次。					
3. 自检测软件无论在哪个系统构件中发现了故障，必须能发出声音报警，泵显示器应当指示故障所在的构件的名字，胰岛素传送因而要暂停。					
4. 系统需要插入一个干预系统，允许系统用户修改通过计算得到、将由系统传送的胰岛素剂量。					
5. 干预的量一定不能大于预设值（最大干预量），这是医务人员对系统事先设置好的。					

图 12-7 简化的危险日志条目

安全评审包括评审软件规格说明、软件设计以及源代码，其目的是发现软件潜在的危险状态，这些不是全自动化的过程，它需要人细心检查可能影响系统安全的错误：已经造成的错误，假想的错误，忽视的错误。例如，在前面介绍的飞机事故中，一项安全评审可能会质

疑这一假设：飞机在地面上且两轮负重时，两轮在旋转。

安全评审应该由危险日志驱动。对于每项已经确定的危险，评审团队检测系统并判断系统能否安全规避该危险。评审小组报告中提出的任何疑点都需要由系统开发团队解决。第 24 章将讨论不同类型的评审，包括软件质量保证。

软件安全认证用在外部构件引入安全关键系统时。当系统所有的模块都在本地开发时，开发过程使用的所有信息都能被维护。然而，开发一些能直接从其他供应商处购买的构件是不合算的。安全关键系统开发的问题是，这些外部构件可能与本地开发的构件有不同的标准，它们的安全是未知的。

因此，对于外部构件必须要经过认证才能引入内部系统中。需要有一个不同于开发团队的安全认证团队对外部构件进行额外的确认和验证（V&V）。如果有合适的外部构件，他们会与构件开发人员联系，了解他们的开发流程，检测构件源代码。如果安全认证团队认为外部构件满足需求规格说明且没有“隐藏”的功能，他们就允许在安全关键系统中使用该外部构件。



软件工程师许可

在许多工程领域，负责安全的系统工程师必须是经过认证的。无经验的、资质差的工程师是不能对安全负责的。美国 30 个州颁发了一些类型的软件工程师执照，其中涉及了安全相关系统开发，这些州要求参与安全关键系统开发的工程师有执照，且具备中等水平的资格和经验。这是一个有争议的问题，在许多其他国家是不要求执照的。

<http://software-engineering-book.com/safety-licensing/>

12.3.2 形式化验证

正如第 10 章中所讨论的那样，软件开发的形式化方法是基于作为系统规格说明的形式化模型的。这些形式化方法主要关注对规格说明的数学分析，将规格说明转换成一个更详细的、语义上等价的描述，或者使用形式化方法来验证系统的一种描述和另一种描述在语义上是等价的。

确保安全关键系统的安全性一直是软件形式化开发方法的主要驱动力。全面的系统测试异常昂贵，也无法保证覆盖软件的所有故障。对于分布式系统来说不同构件同时运行，更是难以测试。一些安全关键的地铁系统就是在 20 世纪 90 年代用形式化方法开发的（Dehbonei and Mejia 1995；Behm et al. 1999）。如空客公司也用形式化方法开发关键系统（Souyris et al. 2009）。

形式化方法可以在 V&V 过程的不同阶段中使用：

1. 开发系统的形式化规格说明并对其进行数学分析可能是为了一致性。这种技术对发现规格说明中的错误和遗漏是有效的。模型检测是一种特别有效的规格说明分析方法，下一节会讨论它。
2. 使用数学证明可以形式化检验软件系统的代码和它的规格说明是一致的。这就要求有一种形式化的规格说明且对发现程序错误和某些设计错误是有效的。

因为形式化的系统规格说明和程序代码之间在语义上存在较大的差距,所以很难证明一个单独开发的程序是和它的规格说明相一致的。因此,程序验证的工作现在是基于变换开发的。在变换开发过程中,形式化的规格说明通过一系列的表示系统地转换成程序代码。软件工具支持变换开发,并且帮助核实相应表示的一致性。B方法很可能是最为广泛使用的形式化的变换方法(Abrial 2010)。这种方法已经被用于开发火车控制系统和航空电子设备软件。

形式化方法强调使用这些方法能使系统更加可靠与安全。形式化验证表明开发的程序符合它的规格说明并且其实现中的错误不会危害系统的稳定性。如果用CSP(Schneider, 1999)这样的语言写的规格说明来开发并发系统的形式化模型,可能会发现那些导致最终程序死锁的情况,并且能够解决这些问题。这是很难仅仅通过测试来完成的。

然而,形式化规格说明和证明并不能保证软件在实际使用中是可靠的。

1. 规格说明可能没有反映用户和系统利益相关者真正的需求。正如第10章中所提到的,系统利益相关者几乎不可能理解形式化符号系统,所以他们不能通过直接阅读形式化规格说明来找到错误和遗漏。这就意味着形式化的规格说明极有可能包含错误,不是系统需求的精确表示。

2. 证明可能包含错误。程序证明是庞大和复杂的,所以,像庞大和复杂的程序本身一样,它们通常也包含错误。

3. 证明所假设的使用方式可能是错误的。如果系统不按假设的情况使用,那么证明就是无效的。

检验一个重要软件系统需要很多时间,还需要数学专家和专门的工具,如定理证明器。因此这是一个极端昂贵的过程,随着系统规模的增加,形式化验证的花费将不成比例地增加。

很多软件工程师认为形式化验证是不划算的,他们认为对系统的相同水平信任程度可以使用其他验证技术,例如审查和系统测试,而这要便宜得多。然而,一些使用形式化验证方法的公司(例如空客)声称,针对构件的单元测试是不必要的,因为单元测试会极大地增加花费(Moy et al. 2013)。

形式化方法和形式化验证在关键软件系统的开发中扮演着重要的角色。形式化规格说明对发现规格说明中的问题是十分有效的,这些问题通常导致系统失效。尽管形式化验证对于大型系统来说仍然是不现实的,但是它能用来检验那些安全要求极高和信息安全要求极高的构件。

12.3.3 模型检测

使用演绎的方法形式化地验证系统是比较困难且昂贵的,人们又提出了一些形式化分析方法,它们基于一个更有限的正确性概念。这些方法中最为成功的是模型检测(Jhala and Majumdar 2009)。模型检测包括创建一个系统模型,并且使用特殊的软件工具检查模型的正确性。模型检测中的各个阶段如图12-8所示。

这种方法被广泛用来检查硬件系统的设计,并且越来越多地用在像NASA的火星探测车(Regan and Hamilton, 2004)和空客公司的航空电子设备这样的关键软件系统中。

已经开发了很多不同的模型检测工具。SPIN就是一个软件模型检测器的早期例子(Holzmann 2003)。最近的系统包括微软的SLAM(Ball, Levin, and Rajamani 2011)和PRISM(Kwiatkowska, Norman, and Parker 2011)。

模型检测过程包括建立系统形式化模型，这里通常要用到扩展有限状态机。模型检测系统使用什么语言，模型就以什么语言表达。例如，SPIN 模型检测器使用 Promela 语言。找出一组需要的性质并以形式化符号表达，它们都是基于时序逻辑的。例如，对于野外气象站系统，一个性质可能是系统总是从“记录”状态到达“传送”状态。

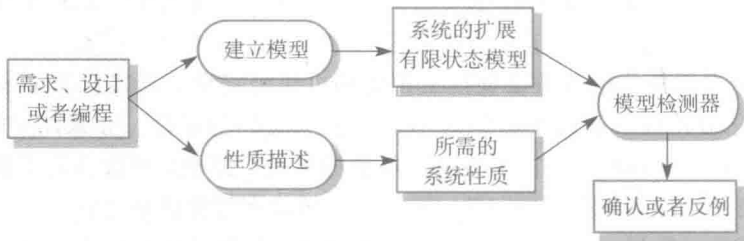


图 12-8 模型检测

模型检测器接着要在模型中探测所有路径（即所有可能的状态转换），检查每条路径所保持的性质。如果成功，模型检测器会确认这个模型关于这个性质是正确的。如果对于某条特殊的路径此性质不能保持，模型检测器就输出一个反例来说明性质在哪里出错了。模型检测在并发系统的有效性验证中特别有用，众所周知，由于并发系统对时间的敏感性很高这种验证非常难以实施。检测器可以通过交叉探测和并发转换来发现潜在问题。

模型检测中的一个关键问题是系统模型的创建。如果模型需要人工建立（从需求或设计文档出发），这将是一个花费很高的过程并且会占用很多时间。另外，也有可能建立的模型不是需求和设计要求的精确模型。因此，如果模型能够从源代码开始自动地建立是最好的选择。有很多现成的直接为 Java、C、C++ 和 Ada 程序工作的模型检测器。

模型检测时计算开销很大，因为它使用穷举方法来检查所有通过系统模型的路径。随着系统规模的增加，状态也随之增加，结果是需要检查的路径数目也增加了。这就意味着，对于大型系统，模型检测可能是不实际的，原因是执行这些检查要耗费大量的计算时间。然而，随着识别部分未探索过的状态的算法的改进，在关键系统开发中使用模型检测变得更加可行。由于这些算法被集成至模型检测器，所以系统常规开发中将更多地应用模型检测器。

12.3.4 静态程序分析

静态程序分析器是一个软件工具，扫描程序源代码，检测可能的缺陷和异常之处。它不需要执行程序，而是通过解析程序文本识别出程序语句的各个部分。由此，分析语句是否合乎规则，推断出程序的控制结构，并计算出可能的程序变量值。静态程序分析器弥补了语言编译器提供的错误检测工具的不足，可以作为审查过程的一部分或者作为一个单独的 V&V 过程活动。

自动程序分析比详细的代码评审更快、成本更低。然而，它不能够发现在程序审查会议中所指出的几类错误。

静态分析工具（Lopes, Vicente, and Silva 2009）使用的是系统源代码，并且至少对于一些类型的分析不需要进一步的输入。这就意味着程序员不需要学习专门的符号来写系统规格说明，因此分析的好处就显而易见了。这让自动静态分析比形式化验证和模型检测更容易引入开发过程中。

自动静态分析的意图是引起代码阅读器对程序中异常的注意，例如变量没有初始化、变量未使用、变量值越界等。由静态分析能检测出来的内容在图 12-9 中列出。

当然，具体的语言需要具体的检查并且取决于该语言的约定。异常通常是程序错误和遗漏的结果，所以它们凸显那些在程序运行时会出现的东西。然而，我们应该明白，这些异常不一定是程序缺陷。它们可能是故意制作的，或者是不会产生不良后果的。

在静态分析器中，可能需要实现 3 个级别的检验：

1. 特有错误检查。在这个级别上，静态分析器知道程序员使用像 Java 或 C 这样的语言编写常见的错误。分析器分析代码，查找出特有的问题并且标记出来呈现给程序员。尽管相对简单，分析是基于常见错误的，但是却是非常符合成本效益的。Zheng 和他的合作者 (Zheng et al. 2006) 研究了在 C 和 C++ 语言的大型代码中使用静态分析，发现 90% 的错误是由 10 种特有错误造成的。

2. 用户定义错误检查。在这种方法中，静态分析器的用户可能定义了错的格式，因此扩展类型的错误可以被检测到。在必须保持顺序的情况下（例如，A 方法必须始终在 B 方法之前被调用），这种方法显得特别有用。久而久之，机构可以收集发生在程序中的常见错误信息，并且扩展静态分析工具来标出这些错误。

3. 断言检查。这是静态分析中最为常规、最强大的方法。开发者在程序中包含形式化断言（通常写成固定格式的注释），以声明在程序某个位置必须满足的关系。例如，断言可能声明一些变量的值必须在 $x \sim y$ 的范围内。分析器符号化地执行代码并突显断言不能保持的语句。

静态分析对查找程序中的错误是很有效的，但是一般会产生大量的“误报”。本来没有错误的代码部分，但是按静态分析器的规则却检查出有错误的可能性。误报的数目可以通过用断言的形式对程序增加更多的信息来减少。当然，这需要代码开发者额外的工作来完成。在代码本身能够被检查出错误之前，对产生误报的筛选工作就要完成。

现在许多组织在软件开发中使用静态分析技术。微软利用静态分析技术检测硬件驱动程序，驱动程序崩溃会产生严重影响。微软在更广泛的范围内寻找安全问题以及影响程序可靠性的错误。检查众所周知的问题如缓冲区溢出是有效的，这样可以提高安全性，攻击者往往基于这些问题进行攻击。攻击可能针对很少使用的代码段，这些代码段未进行过全面测试。静态分析是一种找到这些类型漏洞的符合成本效益的方式。

12.4 安全案例

正如前面所讨论的，许多安全关键的软件密集型系统处于监管之下。外部的认证机构对

缺陷分类	静态分析检查
数据缺陷	未初始化就使用的变量 声明了但从未使用的变量 赋值两次但在两次赋值间未使用的变量 数组越界 未声明的变量
控制缺陷	不可达的代码 不受控制的循环分支
输入 / 输出缺陷	在没有干预赋值的情况下变量输出两次
接口缺陷	参数类型不匹配 参数个数不匹配 没用处的函数结果 未调用的函数和程序
存储管理缺陷	未赋值的指针 指针计算 内存泄漏

图 12-9 自动静态分析检测

于开发和部署有重要影响。政府是监管机构，以确保商业公司不部署对公众、环境以及国际经济有威胁的系统。安全关键系统的拥有者必须相信监管者会尽全力保证他们系统的安全。监管者评估系统的安全案例，并提出证据和论据来说明该系统的正常运行不会对用户造成伤害。

证据会在系统开发过程中收集。它包含危害分析和缓解、测试结果、静态分析，开发过程信息、评审会议记录等。它被组织和包装成一个安全案例，详细地介绍了为什么系统的所有者和开发者相信系统是安全的。

安全案例是一组文档，它包括经过确认的系统描述、开发系统所使用的过程的相关信息，以及更重要的能证明系统安全的逻辑论证。Bishop 和 Bloomfield (Bishop and Bloomfield 1998) 给安全案例下了一个更简洁的定义：

安全案例是一组文档化的证据，它提供了令人信服的和有效的论证，证明在给定的环境下系统对于特定的应用是安全的。[Ⓐ]

安全案例的组成和内容依赖于所要确认的系统的类型和它的操作上下文。图 12-10 给出了一个软件安全案例的可能组成，但是还没有广泛使用的工业标准。安全案例的结构根据行业和领域的成熟度各不相同。例如，原子能安全案例已经使用很多年了。它们非常广泛并且以一种原子能工程师所熟悉的方式呈现。但是，医疗设备的安全案例最近才被引入。它们的结构更加灵活，并且案例本身没有原子能案例具体。

章 节	描 述
系统描述	系统概述和对其关键构件的描述
安全需求	从系统需求规格说明中导出的安全需求。其他系统相关需求的详细内容也会包括在内
危险和风险分析	描述所识别的危险和风险以及所采取的降低风险措施的文档。危险分析和危险日志
设计分析	一组结构化的论证（参见 12.4.1 节），以证明为什么设计是安全的
验证和确认	描述所使用的 V&V 过程，适当的地方还会包括对系统的测试计划。测试结果的总结会给出所检测到且改正了的缺陷。如果使用了形式化方法，会有一个形式化系统描述和对此描述的其他分析。对源代码的静态分析记录
评审报告	对所有设计和安全评审的记录
团队能力	参与安全相关的系统开发和验证的所有团队成员的能力的证据
QA 过程	在系统开发过程中所执行的质量保证过程（参见第 24 章）的记录
变更管理过程	对所提出的所有变更、所执行的活动的记录，有时还包括对这些变更的安全性的判断。配置管理程序和配置管理日志
相关的安全案例	指向其他会对此安全案例产生影响的安全案例

图 12-10 软件安全案例的内容

安全案例是指一个系统作为一个整体，作为该案例的一部分，可能有一个附属的软件安全案例，所以软件安全案例总是更广泛的系统安全案例（展示整个系统的安全）的一部分。当构造软件安全案例时，需要把软件失效同更广泛的系统失效关联起来，说明不会出现这些软件失效，或者是软件失效不会以一种能最终导致系统失效的方式传播。

安全案例都是非常庞大以及复杂的文档，因此，它们难以生产以及维护。由于高昂的成

Ⓐ Bishop, P., and R. E. Bloomfield. 1998. " A Methodology for Safety Case Development. " In Proc. Safety-Critical Systems Symposium. Birmingham, UK: Springer. <http://www.adelard.com/papers/sss98web.pdf>

本,安全关键系统的开发人员必须将安全案例需求加到开发预算当中去。

1. Graydon 等人 (Graydon, Knight, and Strunk 2007) 认为安全案例的开发应该与系统设计和实现紧紧地结合在一起。这就意味着系统设计决策会受安全案例需求的影响。那些可能明显增加案例开发难度和花费的设计选择必须尽量避免。

2. 监管者对于安全案例的可接受程度有他们自己的尺度。因此,让开发人员与监管人员在开发早期接触,了解监管人员对于系统安全的期待,意义重大。

由于要时刻记录以及高费用的综合系统验证和安全保证过程,安全案例的开发成本巨大。系统变更与重做也增加了安全案例的开发成本。当系统有硬件或者软件变更时,大部分安全案例都可能需要重写,以证明系统的安全性没有受到变化的影响。

12.4.1 结构化论证

决定一个系统在使用中是否安全,应该基于逻辑论证。这些论证应该表明所提出的证据能够支持系统信息安全性和可依赖性的断言。这些断言可能是绝对的(事件 X 将会发生或将不会发生),也可能是以一定的概率(事件 Y 发生的概率是 0.n)。论证就是把证据和断言联系起来。如图 12-11 所示,论证是一个关系,是一个什么将会发生(断言)和一组收集到的证据之间的关系。论证主要是解释为什么断言能够从得到的证据中推断出来。断言是关于系统信息安全性和可依赖性的声明。

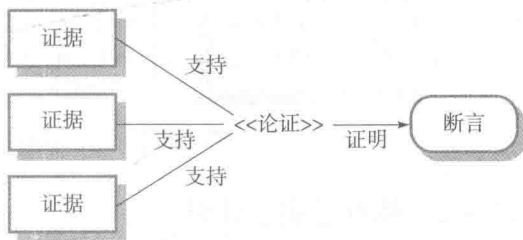


图 12-11 结构化论证

在安全案例中的论证通常称为“基于断言”的论证。在现有证据的基础上,关于系统安全提出了一些断言,论证是解释为什么断言正确的。例如,下面的论证用来证明由控制软件所执行的计算将不会导致给予用户过量的胰岛素的断言。当然,这是一个简化了的论证。一个真正的安全案例中应该给出对证据的详细引用。

断言:胰岛素泵的一次最大量不超过 maxDose,这里 maxDose 是经过核定的对一个特殊病人一次使用的安全剂量。

证据:胰岛素泵软件控制程序的安全论证(12.4.2 节中会讨论安全论证)。

证据:胰岛素的测试数据集。在 400 份测试(提供了全面的代码覆盖)中, currentDose 的值从来没有超过 maxDose。

证据:胰岛素泵控制程序的静态分析报告。这种控制软件的静态分析揭示出没有任何异常影响 currentDose 的值,表示要注射的胰岛素剂量的程序变量也没有任何异常。

论证:给出的安全证据表明可以计算出的胰岛素剂量最大量等于 maxDose。

总之,有充分的信心去合理地假设,证据证明了这样的一个断言,即胰岛素泵不会计算出一个超出最大单次剂量的传输剂量。

注意到证据的表示既冗余又是多样的。软件经过了多个不同机制的检查,这些机制之间存在很大的重叠性。如在第 10 章所讨论的,使用冗余性和多样性的过程增强了信心。如果遗漏和错误不能被一个有效性验证过程检查出来,则很可能被另一个验证过程发现。

通常会有很多关于系统安全性的断言。一个断言的正确性通常取决于其他断言是否是正确的。因此,断言可能被组织成层次的结构。图 12-12 表明了对于胰岛素泵断言层次结构的

一部分。为了证明高层断言是正确的，必须先通过对低层断言的论证。如果可以说明每一低层的断言都是正确的，那么就能够推断出较高层次的断言也是正确的。

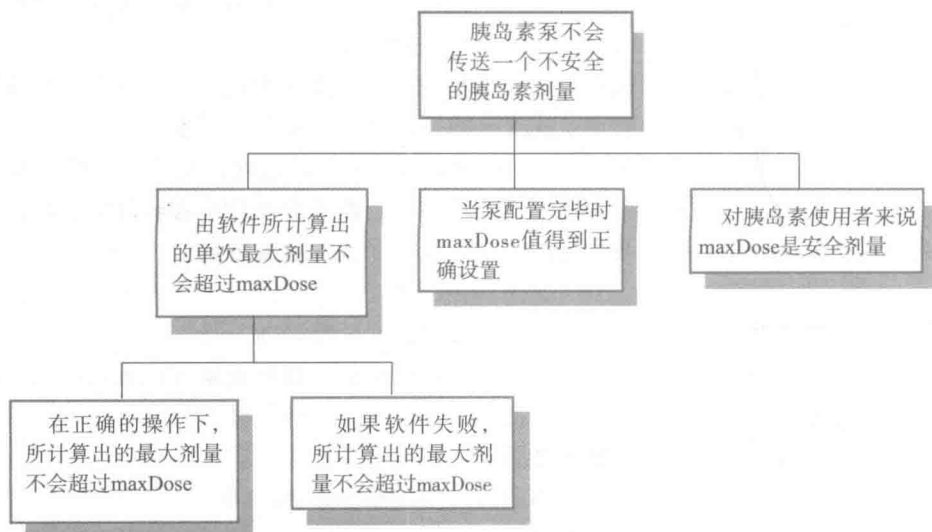


图 12-12 一个胰岛素泵的安全性断言层次结构

12.4.2 软件的安全论证

一个支持系统安全性方面工作的一般假设是，能导致安全性要求极高系统危险的系统错误数目远远小于系统中可能存在的总的错误数目。安全性保证可以集中于这些有潜在危险的错误。如果能够证明这些错误不可能发生，或者是如果发生了，相关的危险也不会导致系统故障，那么这个系统就是安全的。这是软件安全论证的基础。

软件安全论证是一种结构化的论证，说明一个程序达到了它的安全性责任。在安全论证中，并不需要证明程序完全符合系统描述，只需要证明程序的执行不会进入不安全的状态。这就意味着，安全论证比正确性论证花费更少。你不需要去考虑所有的程序状态，可以仅仅将注意力集中在可能导致危险的状态上。

安全论证试图证明，假设运行环境是正常的，那么程序应当是安全的。安全论证通常是基于矛盾的，如果你认为系统是不安全的，那么证明不可能达到一个不安全的状态。在创建安全论证过程中包括以下几个步骤：

1. 首先假设通过危险分析找出来的一个不安全状态能通过程序执行而到达。
2. 写一个谓词（逻辑表达式）来定义此不安全状态。
3. 然后系统地分析系统模型或程序代码，并给出所有能到达此状态的程序路径的所有终止条件，证明这些路径的终止条件与不安全状态谓词是相矛盾的。如果矛盾，对不安全状态的初始假设就是不正确的。
4. 如果这个证明过程对所有识别出来的危险逐个使用过，那就可以证明软件是安全的。

安全论证可以应用在不同的水平上，从需求到设计模型再到程序代码。在需求水平上，我们试图证明没有遗漏安全性需求，并且需求没有对系统做出无效的假设。在设计水平上，我们可能去分析一个系统的状态模型以找出不安全状态。在程序代码水平上，我们通过安全性要求极高的代码来考虑所有的路径，以证明会导致矛盾出现的所有执行路径。

作为一个例子，我们来看图 12-13 给出的胰岛素注射系统中的一段程序代码。这段代码计算出要注射的胰岛素的剂量，然后采用一些安全性检查来降低过量胰岛素被注射的可能性。对这段代码所设计的安全论证需要说明胰岛素的输入剂量不会超过单次剂量的上限水平。这是每个糖尿病患者和他们的医疗顾问相互讨论确定下来的。

```

- The insulin dose to be delivered is a function of
- blood sugar level, the previous dose delivered and
- the time of delivery of the previous dose

currentDose = computeInsulin ();

// Safety check-adjust currentDose if necessary.

// if statement 1
if (previousDose == 0)
{
    if (currentDose > maxDose/2)
        currentDose = maxDose/2 ;
}
else
    if (currentDose > (previousDose * 2) )
        currentDose = previousDose * 2 ;

// if statement 2

if ( currentDose < minimumDose )
    currentDose = 0 ;
else if ( currentDose > maxDose )
    currentDose = maxDose ;
administerInsulin (currentDose) ;

```

图 12-13 有安全性检查的胰岛素剂量计算

为了证明安全性，不必证明系统注射了正确剂量，只要它没有给病人注射过量就可以了。你的工作是建立在假设 maxDose 对于系统用户来说是个安全水平的基础上的。

为了构造安全论证，要找出一个定义不安全状态的谓词，这里是 $\text{currentDose} > \text{maxDose}$ 。接下来就要证明所有的程序路径都与这个不安全断言相矛盾。如果是这样，这个不安全条件就不可能是真实的。如果能证明出矛盾，就可以很有信心地说程序不会计算出不安全的胰岛素剂量。可以以图形的形式构造和表示安全论证，如图 12-14 所示。

图 12-14 所示的安全论证中，有 3 个可能的程序路径可以引起 `administerInsulin` 方法调用。我们必须说明所传输的胰岛素剂量不会超过 maxDose 。所有到达 `administerInsulin` 的程序路径包括：

1. if 语句 2 的两个分支都没有执行，这只能发生在 currentDose 大于等于 minimumDose 和小于等于 maxDose 时。这是后置条件：

$\text{currentDose} \geq \text{minimumDose}$ and $\text{currentDose} \leq \text{maxDose}$

2. if 语句 2 的 then 分支被执行，这时对 currentDose 赋值 0 的语句执行。因此，它的花后置条件是 $\text{currentDose} = 0$ 。

3. if 语句 2 的 else-if 分支被执行，这时对 currentDose 赋值 maxDose 的语句得到执行。因此，在这条语句被执行后，我们知道后置条件是 $\text{currentDose} = \text{maxDose}$ 。

在所有的情况中；后置条件都与不安全状态的前置条件 ($\text{currentDose} > \text{maxDose}$) 是矛盾的。因此我们断言初始假设是不正确的，计算是安全的。

为了构造结构化的论证，使程序不会做出不安全的计算，首先要通过代码确定能够通向

潜在的不安全状态的所有可能路径。然后由这些不安全状态出发，反向工作，考虑通向不安全状态的每个路径的所有状态变量的最后一次赋值。如果能够说明这些变量中没有一个赋值是不安全的，那么就表示最初的假设（计算是不安全的）是不正确的。

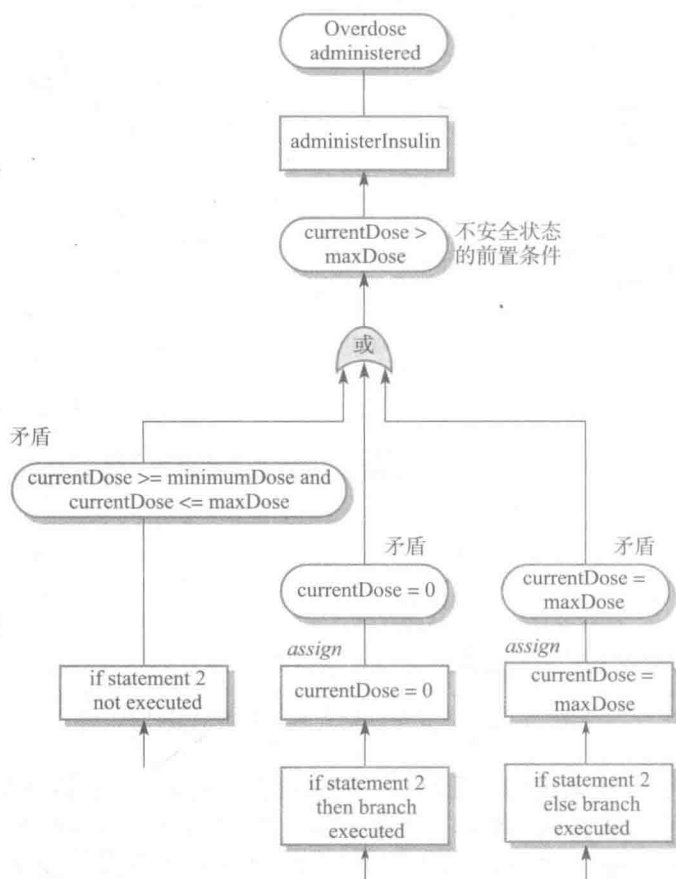


图 12-14 基于矛盾证明的非形式化安全论证

反向工作很重要，因为它表示你能够忽略所有的中间状态，只考虑通向代码出口条件的那些最终状态。之前的值对于系统的安全性已经无关紧要。在这个例子中，所有需要考虑的是在 `administerInsulin` 方法执行之前对 `currentDose` 的最近一次可能的赋值。在安全论证中可以忽略一些计算，如图 12-13 中的 `if` 语句 1，因为它们的结果会被之后的语句所覆盖。

要点

- 安全关键系统是故障可能导致人员伤害或死亡的系统。
- 危险驱动的方法可以用在理解安全关键系统的安全需求上。要找出潜在的危险并且分解这些危险（使用如故障树分析的方法），以发现它们的根本原因。然后要描述需求来避免出现问题或者从问题中恢复。
- 有一个定义完好的并经过认证的开发过程对安全关键系统开发非常重要。这样的过程一定要包括对潜在危险的识别和监控。
- 静态分析是 V&V 的一种检查系统源代码（或其他表示）的方法，用于查找错误和异

常。它检查程序的所有部分，而不仅仅是那些会被测试的部分。

- 模型检测是静态分析的一种形式化方法，它彻底检查系统的所有状态以发现潜在的错误。
- 安全和可依赖性案例搜集所有能表明系统安全和可靠的证据。当外部监管者在系统使用前对系统进行验证时需要用到安全案例。

阅读推荐

《Software: System Safety and Computers》是一本针对安全关键系统的各个方面展开彻底讨论的书。它的长处在于对危险分析的阐述和从这些分析中导出需求。(N. Leveson, Addison-Wesley, 1995)

《Safety-Critical Software》是《IEEE Software》杂志的一期专刊，专注于安全关键系统。它包括基于模型的安全关键系统的开发、模型检测和形式化方法。(IEEE Software, 30(3), May/June 2013)

《Constructing Safety Assurance Cases for Medical Devices》介绍了一个如何为镇痛泵创建安全案例的实例。(A. Ray and R. Cleaveland, Proc. Workshop on Assurance Cases for Software-Intensive Systems, San Francisco, 2013) <http://dx.doi.org/10.1109/ASSURE.2013.6614270>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap12/>

支持视频的链接: <http://software-engineering-book.com/videos/reliability-and-safety/>

练习

- 12.1 指出 6 个客户产品，它们可能由安全关键软件系统控制。
- 12.2 为什么图 12-3 所示的风险三角形的边界容易随着时间的推移和变化了的社会意识而发生改变？
- 12.3 在胰岛素泵系统中，用户需要定期更换针头和胰岛素的供给，还会改变单次剂量的最大值和日剂量的最大值。提出 3 个可能发生的用户错误，并给出安全需求来避免这些错误造成事故。
- 12.4 一个治疗癌症患者的安全关键软件系统有两个主要的构件：
 - 一个辐射治疗仪，能对肿瘤部位进行辐射治疗，辐射剂量能够控制。这个仪器是由一个嵌入式软件系统控制的。
 - 一个治疗数据库，包括对每位病人治疗的详细情况。治疗需求被输入到这个数据库中，并自动输出到辐射治疗仪中。

识别出 3 个可能出现在系统中的危险。对每个危险，提出能减少危险造成事故可能性的防范需求。解释为什么你提议的防范措施能减少危险的发生。

- 12.5 当超过轨道限速或者火车驶入一个当前红灯的轨道段（例如，当前轨道不应该驶入）时，火车保护系统能自动停止火车。这个火车保护系统有两个安全关键需求：
 - 火车不应该驶入信号灯为红灯的轨道段。
 - 火车不会超过轨道的限速。

假设轨道段的信号状态和速度限制在进入轨道段之前就被传输到列车的车载软件，请从系统安全需求的角度，为车载软件提出 5 个可能的功能性系统需求。

- 12.6 解释一下在软件系统开发中何时使用形式化描述和检验是划算的。你为什么认为安全关键系统的工程师不赞成使用形式化方法？
- 12.7 为何使用模型检测比常规方法验证程序正确性更加实惠？
- 12.8 列出需要系统软件安全案例的 4 种系统类型。解释为什么需要安全案例。
- 12.9 核废料存储设施中的门锁控制机制是为了安全操作。它能保证只有在辐射防护罩保护或辐射水平降低到给定值 (dangerLevel) 的情况下方可进入，这就是说：
- (i) 远程控制的辐射防护罩安装在房间里面，门可以由经授权的操作人员开启。
 - (ii) 如果房间里的辐射水平低于给定值，门可以由经授权的操作人员开启。
 - (iii) 经授权的操作人员是通过输入授权的入口密码确认的。

图 12-15 中给出的是用来控制门锁机制的程序段。注意这里安全状态是入口不应该允许进入。使用本章中讨论的方法，给出此代码的一个安全论证。使用行数指定特定的语句。如果你发现代码是不安全的，对如何修改该代码使之安全给出建议。

```

1      entryCode = lock.getEntryCode ();
2      if (entryCode == lock.authorizedCode)
3      {
4          shieldStatus = Shield.getStatus ();
5          radiationLevel = RadSensor.get ();
6          if (radiationLevel < dangerLevel)
7              state = safe;
8          else
9              state = unsafe;
10         if (shieldStatus == Shield.inPlace())
11             state = safe;
12         if (state == safe)
13         {
14             Door.locked = false ;
15             Door.unlock ();
16         }
17         else
18         {
19             Door.lock ( );
20             Door.locked := true ;
21         }
22     }

```

图 12-15 门禁代码

- 12.10 从事安全性相关系统的需求描述和开发工作的软件工程师需要某种专业认证吗？解释你的理由。

参考文献

- Abrial, J. R. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.
- Ball, T., V. Levin, and S. K. Rajamani. 2011. "A Decade of Software Model Checking with SLAM." *Communications of the ACM* 54 (7) (July 1): 68. doi:10.1145/1965724.1965743.
- Behm, P., P. Benoit, A. Faivre, and J.-M. Meynadier. 1999. "Meteor: A Successful Application of B in a Large Project." In *Formal Methods'99*, 369–387. Berlin: Springer-Verlag. doi:10.1007/3-540-48119-2_22.

- Bishop, P., and R. E. Bloomfield. 1998. "A Methodology for Safety Case Development." In *Proc. Safety-Critical Systems Symposium*. Birmingham, UK: Springer. <http://www.adelard.com/papers/sss98web.pdf>
- Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.
- Dehbonei, B., and F. Mejia. 1995. "Formal Development of Safety-Critical Software Systems in Railway Signalling." In *Applications of Formal Methods*, edited by M. Hinchey and J. P. Bowen, 227–252. London: Prentice-Hall.
- Graydon, P. J., J. C. Knight, and E. A. Strunk. 2007. "Assurance Based Development of Critical Systems." In *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, 347–357. Edinburgh, Scotland. doi:10.1109/DSN.2007.17.
- Holzmann, G. J. 2014. "Mars Code." *Comm ACM* 57 (2): 64–73. doi:10.1145/2560217.2560218.
- Jhala, R., and R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4). doi:10.1145/1592434.1592438.
- Kwiatkowska, M., G. Norman, and D. Parker. 2011. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In *Proc. 23rd Int. Conf. on Computer Aided Verification*, 585–591. Snowbird, UT: Springer-Verlag. doi:10.1007/978-3-642-22110-1_47.
- Leveson, N. G., S. S. Cha, and T. J. Shimeall. 1991. "Safety Verification of Ada Programs Using Software Fault Trees." *IEEE Software* 8 (4): 48–59. doi:10.1109/52.300036.
- Lopes, R., D. Vicente, and N. Silva. 2009. "Static Analysis Tools, a Practical Approach for Safety-Critical Software Verification." In *Proceedings of DASIA 2009 Data Systems in Aerospace*. Noordwijk, Netherlands: European Space Agency.
- Lutz, R. R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In *RE'93*, 126–133. San Diego, CA: IEEE. doi:10.1109/ISRE.1993.324825.
- Moy, Y., E. Ledinot, H. Delseny, V. Wiels, and B. Monate. 2013. "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience." *IEEE Software* 30 (3) (May 1): 50–57. doi:10.1109/MS.2013.43.
- Perrow, C. 1984. *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books.
- Regan, P., and S. Hamilton. 2004. "NASA's Mission Reliable." *IEEE Computer* 37 (1): 59–68. doi:10.1109/MC.2004.1260727.
- Schneider, S. 1999. *Concurrent and Real-Time Systems: The CSP Approach*. Chichester, UK: John Wiley & Sons.
- Souyris, J., V. Weils, D. Delmas, and H. Delseny. 2009. "Formal Verification of Avionics Software Products." In *Formal Methods' 09: Proceedings of the 2nd World Congress on Formal Methods*, 532–546. Springer-Verlag. doi:10.1007/978-3-642-05089-3_34.
- Storey, N. 1996. *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.
- Veras, P. C., E. Villani, A. M. Ambrosio, N. Silva, M. Vieira, and H. Madeira. 2010. "Errors in Space Software Requirements: A Field Study and Application Scenarios." In *21st Int. Symp. on Software Reliability Engineering*. San Jose, CA. doi:10.1109/ISSRE.2010.37.
- Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. "On the Value of Static Analysis for Fault Detection in Software." *IEEE Trans. on Software Eng.* 32 (4): 240–253. doi:10.1109/TSE.2006.38.

信息安全工程

目标

本章的目的是介绍开发应用系统时所需要考虑的诸多信息安全问题。阅读完本章后，你将：

- 理解信息安全工程的重要性，以及应用信息安全与基础设施信息安全的区别；
- 了解如何用基于风险的方法获取信息安全需求及分析系统设计；
- 了解软件的体系结构模式和针对信息安全系统工程的设计指导方针；
- 理解为何信息安全测试和保证是困难且昂贵的。

20 世纪 90 年代互联网的广泛使用给软件工程师带来了新的挑战，即需要设计和实现能保证信息安全的系统。随着越来越多的系统与互联网连接，各种各样的外部攻击被人为地设计，严重威胁着系统的安全。开发可依赖系统的问题在急剧增加。系统工程师不得不考虑来自怀有恶意的攻击者和技术能手攻击者这两类攻击者的威胁，也要考虑开发过程中由一些意外人为错误所导致的众多问题。

现在看来，设计能抵御外部攻击并在遭受攻击后成功恢复的系统是十分必要的。没有对信息安全的预防，网络上的系统不可避免地会受到黑客的攻击。攻击者可能滥用系统硬件、盗窃机密数据或者是破坏系统所提供的正常服务。

在信息安全系统工程中，必须考虑以下 3 个信息安全的维度。

1. 机密性。系统中机密的信息可能被泄露，被未获授权的人或程序访问到。例如，从电子商务系统窃取信用卡数据是一个机密性问题。

2. 完整性。系统中的完整性信息可能被损坏或引起错误，使其异常或不可靠。例如，在系统中删除数据的蠕虫是一个完整性问题。

3. 可用性。可用的系统或数据有时可能会无法访问。使服务器过载的拒绝服务攻击是系统可用性受到破坏的一个例子。

这些维度是密切相关的。如果一次攻击使得系统不可用，那么将不能实时更新信息。这意味着系统的完整性可能受到了损害。如果一次攻击成功并且完整性受到损害，那么系统必须停止使用并修复问题。因而，系统的可用性就降低了。

从组织的角度来看，信息安全必须考虑 3 个层次。

1. 基础设施信息安全。涉及所有系统和网络的信息安全维护，它们为组织提供基础设施和一套共享服务。

2. 应用信息安全。涉及个人应用系统或相关系统群的信息安全。

3. 操作信息安全。与组织的系统的安全运行和使用有关。

图 13-1 是一个应用程序系统堆栈的图表，它显示了应用系统在运行时如何依赖于其他系统的基础设施。底层的基础设施是硬件，但应用系统的软件基础设施可能包括：

- 操作系统平台，例如 Linux 或者是各种 Windows 操作系统；
- 系统上运行的其他通用应用，如 Web 浏览器和电子邮件客户端程序；

- 数据库管理系统；
- 支持分布式计算和数据库访问的中间件；
- 用于应用软件开发的可复用构件库。

网络系统是软件控制的，并且网络可能容易遭受信息安全威胁，攻击者可以拦截及读取或更改网络包。然而，这需要专门的设备，所以大多数的信息安全攻击是针对系统的软件基础设施。攻击者专注于软件基础设施，是因为基础设施构件（如 Web 浏览器）被广泛使用。攻击者可以探测这些系统的弱点和共享他们发现的漏洞信息。由于许多人使用相同的软件，这种攻击具有广泛的适用性。

应用	
可复用的构件和库	
中间件	
数据库管理	
通用的共享应用（浏览器、电子邮件系统等）	
操作系统	
网络	计算机硬件

图 13-1 信息安全可能涉及的系统层级

基础设施信息安全主要是一个系统管理问题，系统管理员配置基础设施以抵御攻击。系统信息安全管理包括用户和权限管理，系统软件部署和维护，以及攻击监控、检测和恢复等一系列活动。

1. 用户和权限管理。包括向系统添加和删除用户，确保系统在需要的地方采用了适当的用户认证机制，设置系统的权限让用户只能访问到他们应该访问的那些资源。
2. 系统软件部署和维护。包括系统软件和中间件的安装，对这些系统进行正确配置以保证安全漏洞不会出现；也包括不断地用新版本或补丁更新系统，及时修补发现的安全问题。
3. 攻击监控、检测和恢复。包括对未授权的系统访问的监控、检测攻击和抵御攻击策略落实到位，以及备份程序和数据以保证系统能在遭受外部攻击后恢复正常运行。

运行信息安全主要是一个人类和社会问题。它的重点是确保使用该系统的人不会做出伤害系统信息安全性的行为。例如，用户在自己不在场的时候让系统处于登录状态，这样攻击者就可以很容易地访问系统。用户经常采用不安全的方式来更有效地完成工作，他们这样做有他们自己的合理理由。运行信息安全的一个挑战是提高对信息安全问题的认识，并找到安全性和系统效率之间的正确平衡。

网络安全（cybersecurity）这个术语现在通常用于讨论系统的信息安全性。网络安全是一个广泛的术语，涵盖了公民、企业和关键基础设施保护的所有方面，使他们不受到来自计算机和互联网的威胁。网络安全的范围包括所有系统层次，从硬件和网络经过应用系统到用于访问这些系统的移动设备。本书将在第 14 章讨论一般的网络安全问题，包括基础设施信息安全和韧性工程。

本章专注于应用信息安全工程问题：信息安全需求、信息安全设计、信息安全测试。本章没有讨论通用的信息安全技术，例如加密、访问控制机制或攻击载体（如病毒和蠕虫）。计算机安全的通用教材（Pfleegeer and Pfleegeer 2007；Anderson 2008；Stallings and Brown 2012）中会详细讨论这些技术。

13.1 信息安全和可依赖性

信息安全性是一个反映系统保护自己免受内外部攻击的能力的系统属性。这些外部攻击的可能原因是，大多数一般用途的计算机和移动设备现在都已经联网并且因此可以被外界访问。攻击的例子通常包括：病毒或者木马的安装，系统服务未经许可的使用，系统或数据未经许可的修改等。

如果你真的想要一个尽可能安全的系统,那么最好不要连接互联网。这样,信息安全问题就只需要确保授权用户不会滥用系统,以及限制类似 USB 这样的设备的使用。然而在实践中,对于大多数系统来说连接网络的好处是巨大的,所以与互联网断开连接是不可行的。

在有些系统中,信息安全性是可依赖性中最为重要的内容。军事系统、电子商务系统以及机要信息处理和交换系统都需要具有很高的信息安全等级。比方说,没有机票预订系统是很不方便的,会耽误机票出售。不过,如果系统是不安全的,攻击者可能删除所有的订票信息,那么日常的航班运行将无法继续。

与可依赖性的其他方面一样,有一个与信息安全性有关的专门术语集(Pfleeger and Pfleeger 2007)。这个术语如图 13-2 所示。图 13-3 用来自 Mentcare 系统的信息安全故事来说明这些术语。图 13-4 说明了图 13-2 中定义的信息安全概念如何应用到这个信息安全故事中。

术 语	定 义
资产	必须保护的某种有价值的东西。资产可以是软件系统本身或者系统使用的数据
攻击	利用系统漏洞,攻击者有目的地对系统资产或财物造成损害。攻击可能是外部攻击,也可能是内部攻击
控制	一种减少系统漏洞的保护措施。加密是一个控制的例子,减少了弱访问控制系统的漏洞
暴露	可能对计算机系统造成的损失或损害。这可能是针对数据的损失或损坏,也可能是当信息安全漏洞必须修复时花费的时间和工作量
威胁	有可能(潜在)造成损失或损害的情况。你可以把威胁当作一个受到攻击的系统漏洞
漏洞	基于计算机的系统弱点,可能利用漏洞造成损失或损害

图 13-2 信息安全术语

<p>未经授权访问的 Mentcare 系统</p> <p>诊所工作人员使用用户名和密码登录 Mentcare 系统。该系统要求密码至少 8 个字符长,但允许任何密码设置都无须进一步的检查。一名罪犯发现一位高薪体育明星正在接受心理健康问题的治疗。他想获得非法访问该系统的信息,以便勒索这位明星。</p> <p>通过假扮一名相关的亲属,他在心理健康诊所与护士交谈,发现了如何访问系统以及关于这名护士和他们家人的个人信息。通过检查姓名证章,他发现一些可以访问系统的人的名单。然后,他试图使用这些名字登录系统,并系统性地猜测可能的密码,比如护士子女的名字。</p>	
--	--

图 13-3 关于 Mentcare 系统的信息安全故事

术 语	例 子
资产	每位将接受或已接受治疗病人的记录
攻击	扮演已授权用户
控制	一个密码校验系统,不允许用户密码是常见的名字或字典里常见的单词
暴露	潜在的经济损失来自于未来不想来治疗的病人,因为他们不信任该诊所保留他们的数据。由于体育明星的法律行动而导致的经济损失,声誉受损
威胁	一名未经授权的用户通过猜测另一名授权用户的凭证(登录名和密码)来获取对系统的访问权限
漏洞	身份验证是基于不要求强口令的密码系统,用户可以设置很容易被猜到的密码

图 13-4 信息安全术语示例

系统漏洞可能因为需求问题、设计问题或实现问题而出现,或者由人类、社会或组织的缺陷造成。人们也许会选择易于猜测的密码或在他们可以找到的地方写下密码。系统管理员在设置访问控制或配置文件时出错,用户不安装或不使用保护软件。然而,我们不能简单地把这些问题列为人为错误。用户的错误或遗漏经常反映出不良的系统设计决策,例如,频繁的密码更改(因此用户要写下他们的密码)或复杂的配置机制。

可能出现4种信息安全威胁:

1. 拦截威胁。即攻击者获得对某些资产的访问。因此,对于Mentcare系统一个可能的威胁是攻击者获得了对某位患者的就诊记录的访问权限。

2. 中断威胁。即攻击者使得系统的一部分不可用。所以,一个可能的中断威胁是对于系统数据库服务器的拒绝服务攻击。

3. 修改威胁。即攻击者篡改系统资产。在Mentcare系统中,一个可能的修改威胁是攻击者更改或破坏病人的记录信息。

4. 伪造威胁。即攻击者向系统中插入错误信息。这在Mentcare系统中可能算不上是一个严重的威胁,但是对于银行系统来说绝对是个巨大威胁,将钱转入犯罪者的账户这样的虚假交易可能被加入系统中。

增强系统信息安全性的控制手段基于规避、检测和恢复这三种基本思想:

1. 漏洞规避。用来确保攻击无法成功的控制方法。这里使用的策略是,设计系统使信息安全性问题得到避免。比如说,敏感的军用系统不连接到公用网络,这样外部访问就会很困难。同样,应该考虑使用加密控制手段。任何未经授权的攻击者不能对加密文件进行访问。在现实中,破解强化的加密文件是非常昂贵并且费时的。

2. 攻击检测和压制。控制的意图在于检测和压制攻击。这些控制包括在系统中加入监控系统运行并检查异常活动模式的功能。如果检测到这些攻击,接下来可能会采取行动,比如关闭部分系统,限制某些用户的访问,等等。

3. 暴露限制与恢复。支持从问题中恢复的控制方法。这些控制可以是自动备份的策略和信息“镜像”,也可以是保险措施,弥补一次成功攻击系统所造成的损失。

信息安全与可依赖性的其他属性——可靠性、可用性、安全性和韧性密切相关。

1. 信息安全和可靠性。如果一个系统受到攻击,作为攻击的后果,系统或它的数据被破坏,那么,这可能会导致系统失效,危及系统的可靠性。

系统开发中的错误可能会导致信息安全漏洞。如果一个系统不拒绝意外的输入或不检查数组边界,那么攻击者可以利用这些弱点获得对系统的访问。例如,未检查输入的有效性可能意味着攻击者可以注入并执行恶意代码。

2. 信息安全和可用性。基于Web系统的常见攻击是拒绝服务攻击,其中Web服务器被淹没在来自一系列不同来源的服务请求中。这种攻击的目的是使系统不可用。这种攻击的一个变种是用这种类型的攻击威胁一个网站,以求支付赎金给攻击者。

3. 信息安全和安全性。再次重申,关键问题是破坏系统和它的数据的攻击。安全检查基于如下假设:我们能够分析安全关键软件的源代码,执行代码是对源代码完全准确的翻译。如果不是这样的话,因为攻击者改变了执行代码,可能会诱发安全相关的失效以及导致为软件所做的安全预案失效。

和安全性一样,我们不能给系统的信息安全性分配一个数值,也不能为追求信息安全而穷举测试系统。安全性和信息安全可以被认为具有一种“负面的”或“不应该”的特性,因

为它们涉及那些不该发生的事。由于我们无法证明负面，所以无法证明一个系统是安全的或信息安全的。

4. 信息安全和韧性。韧性将在第 14 章中介绍，它是一个系统特性，反映了抵抗损害事件并从损害事件中恢复的能力。在网络化软件系统中最可能的损害事件是某些网络攻击，所以目前在韧性方面所做的大部分工作是防止、检测并从攻击中恢复过来。

如果我们要创建可靠、可用和安全的软件密集型系统，就必须维护信息安全。它不是附加的——可以之后再添加，而是必须在开发生命周期从早期需求到系统运行的所有阶段中考虑信息安全性。

13.2 信息安全和组织

建立信息安全的系统是昂贵和不确定的。对信息安全失效的成本进行预测是不可能的，所以公司和其他组织发现很难决定应该在系统信息安全方面花多少钱。在这方面，信息安全和安全性是不同的。有法律规定工作场所和运营的安全，安全关键系统的开发者必须遵守这些法律而不去考虑成本。如果他们使用一个不安全的系统，可能会受到法律诉讼。然而，除非一个信息安全失效泄露了个人信息，否则没有法律去防止部署无法保障信息安全的系统。

公司评估风险和损失，这些风险和损失可能来自于对系统资产的某种类型的攻击。然后，他们可能认为接受这些风险更合算，而不是建立一个可以阻止或击退外部攻击的信息安全系统。信用卡公司应用这种方法进行欺诈预防。人们通常可能采用新技术来减少信用卡诈骗。然而，对于这些公司来说，补偿因欺诈而受到损失的用户，比购买和部署预防欺诈的技术更便宜。

因此信息安全风险管理是一种业务而不是技术问题。必须考虑一个成功的系统攻击带来的经济和声誉损失，同时也要考虑为减少损失而部署的信息安全程序和技术成本。为了让风险管理有效，组织应该有一个文档化的信息安全策略：

1. 必须保护的资产。将严格的信息安全程序应用于所有的组织资产没有意义。许多资产是不保密的，公司可以通过免费提供资产来提高其形象。保持公共领域信息安全的成本远低于保持机密信息安全的成本。

2. 不同类型资产所要求的保护水平。并非所有的资产都需要相同的保护水平。在某些情况下（例如，敏感的个人信息），高水平的信息安全是必需的；对于其他信息，损失的后果可能是次要的，所以较低的信息安全水平足够了。因此，一些信息可以提供给任何授权和登录用户，其他信息可能更敏感，只有特定角色或职位的用户可用。

3. 个人用户、管理者和组织的职责。信息安全策略应该列出对用户的期望——例如，使用强密码，注销计算机，锁好办公室。它还定义了用户可以从公司得到什么，比如备份和信息归档服务，以及设备供应。

4. 现有的信息安全程序和技术。出于实用性和成本的原因，继续使用现有的技术来保证信息安全是必不可少的，即使这些技术存在一些已知的局限性。例如，公司可能需要使用登录名/密码进行身份验证，因为其他方法很可能被用户拒绝。

信息安全策略经常设置通用信息访问策略，它应该在整个组织适用。例如，访问策略可以基于访问信息的人的权限或资历。因此，军事信息安全策略可能是：“阅读者仅可以查阅那些文档等级与阅读者的审查等级相同或者低于阅读者的审查等级的文档。”这意味着如果一个阅读者的审查等级为“绝密”，他可以访问标为“绝密”“机密”或者“公开”的文档，

但是不可以访问标记为“最高机密”的文档。

信息安全策略的要点是告知组织中涉及信息安全的每个人，所以不应该是过长的详细说明的技术文档。从信息安全工程的角度来看，信息安全策略从广义上定义了该组织的信息安全目标。信息安全工程关注实现这些目标。

13.2.1 信息安全风险评估

信息安全风险评估和管理是组织活动，专注于识别和理解组织中信息资产（系统和数据）的风险。原则上，各项风险评估应该对所有资产进行，然而，在实践中这可能是不切实际的——如果大量的现有系统和数据库需要进行评估的话。在这种情况下，通用的评估可能适用于所有的系统和数据库。然而，各项风险评估应当针对新系统进行。

风险评估和管理是组织活动，而不是属于软件开发生命周期的一部分的技术活动。其原因是一些类型的攻击不是基于技术的，而是针对更通用的组织信息安全的弱点。例如，攻击者可以通过假装是一个可信任的工程师获取对设备的访问。如果组织有专门的过程来通过设备供应商检查每次访问是否都是事先计划的，那么就可以阻止这种类型的攻击。这种方法比试图使用技术解决问题的方法要简单得多。

当要开发新的系统时，信息安全风险评估和管理应该是一个持续的过程，贯穿从最初的规格说明到运营使用的整个开发生命周期。风险评估的步骤是：

1. 初步风险评估。初步风险评估的目的是识别适用于该系统的通用风险并决定是否可以在合理的成本下达到适当的信息安全等级。在这个阶段，没有详细的系统要求、系统设计或实施技术的决定。你不知道潜在的技术漏洞，或包括在复用系统构件或中间件中的控制。因此，风险评估应该侧重于识别和分析系统的高层次风险。风险评估过程的结果是用来帮助识别信息安全需求的。

2. 设计风险评估。风险评估发生在整个系统开发过程中，可以借助系统设计和实现的技术决策进行告知。评估的结果可能会改变信息安全需求和增加新的需求。识别出已知和潜在的漏洞，并用于系统的实现、测试和部署决策。

3. 运行风险评估。风险评估过程侧重于系统的使用和可能出现的风险。例如，当系统在一个常被中断的环境中时，一种信息安全风险是已登录的用户离开了，计算机无人值守，无人解决出现的问题。为了应对这种风险，可以指定一个超时要求，这样用户在一段时间不活动后可以自动注销。

运行风险评估在系统部署和投入使用之后应该继续。在系统定义时做出的系统运行需求假设可能是不正确的。机构的改变可能意味着系统以不同于原计划的方式使用。因此，运行风险评估会产生新的信息安全需求，这些需求必须在系统升级时实现。

13.3 信息安全需求

系统的信息安全需求描述与安全需求有很多相似之处。对它们进行量化描述是不太现实的，信息安全需求往往是“不应该”需求类型，它定义那些无法接受的系统行为，而不是定义系统功能。

但是，信息安全性比安全性更有挑战，有以下几个原因：

1. 当考虑到安全性的时候，可以认为系统所安装的环境不是有敌意的。没有人想要引起一个安全相关的事故。当考虑到信息安全性时候，必须要假设对系统的攻击是故意的，并

且系统的攻击者了解系统的弱点。

2. 当系统失效并带来安全性的风险时, 需要寻找导致失效的错误或者疏忽。当故意的攻击导致系统失效时, 找到根本原因可能更加困难, 因为攻击者可能会试图掩盖系统失效的原因。

3. 通常关闭系统或者降低系统服务可以避免安全相关的失效, 这些做法是可以接受的。然而, 对系统的攻击可能是拒绝服务攻击, 目的就是要破坏系统的可用性。关闭了系统就意味着攻击成功。

4. 安全相关事件不是由聪明的对手制造的。而攻击者可以在一系列攻击中试探系统的防御, 在他更加了解系统以及系统的反应的时候可以改进攻击的方式。

这些区别意味着信息安全需求的成本一定要比安全需求的成本高。安全需求导致产生功能性系统需求, 需要提供对可能引发系统安全相关失效的事件和缺陷的保护。它们更多关注检查问题并在问题发生时采取行动。相反, 有很多信息安全相关的需求覆盖了系统所面临的各种威胁。

Firesmith (Firesmith, 2003) 找出了包含在系统规格说明中的 10 类信息安全需求:

1. 身份验证需求定义系统是否应该在用户与之交互之前辨认其身份;
2. 认证需求定义系统如何验证用户身份;
3. 授权需求定义合法用户的权力和访问许可;
4. 免疫需求定义系统如何保护自己免受病毒、蠕虫以及类似威胁的入侵;
5. 完整性需求定义如何避免数据损坏;
6. 入侵保护需求定义在系统中应该使用什么机制来检测对系统的攻击;
7. 不可抵赖需求定义参与交易的一方不能对自己已经做出的交易抵赖;
8. 隐私需求定义如何维护数据的私密性;
9. 信息安全的审计需求定义如何对系统的使用进行审计和检查;
10. 系统维护的信息安全需求定义应用如何避免因信息安全机制的意外失效所导致的授权的修改。

当然, 你不会在每个系统中看到所有这些种类的需求。特定的需求取决于特定的系统、特定的使用环境以及特定的用户。

初步风险评估和分析的目的是识别系统和相关数据的通用信息安全风险。这种风险评估对信息安全需求工程过程是一个重要的输入。可以提出信息安全需求来支持通用的风险管理规避、检测和缓解策略。

1. 风险规避需求列出了在设计系统时应避免的风险, 以便使这些风险根本不可能出现。
2. 风险检测需求定义机制来识别出现的风险, 并在损失发生之前中和风险。
3. 风险缓解需求阐述了系统应该如何设计, 以便它可以恢复和修复一些损失发生后的系统资产。

风险驱动的信息安全需求过程在图 13-5 中给出。此过程中的阶段有:

1. 资产识别。识别出可能需要保护的系统资产。系统自身或者是特殊的系统功能以及系统相关的数据都是资产。
2. 资产价值评估。估计你所识别的资产的价值。
3. 暴露评估。也就是评估与每一份资产相关联的潜在损失。这个阶段应该考虑到如信息失窃一类的直接损失、恢复的花费, 以及可能的声誉上的损失。

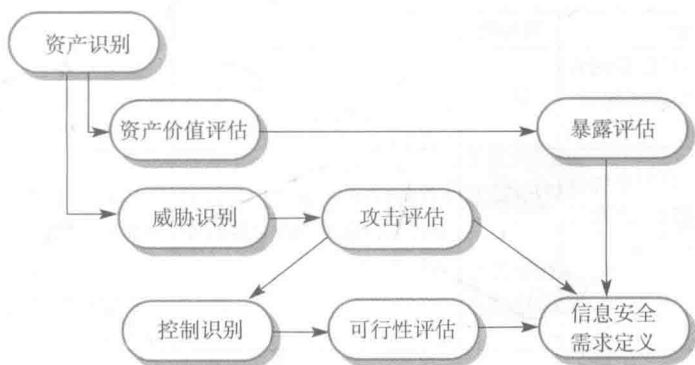


图 13-5 信息安全需求的初步风险评估过程

4. 威胁识别。就是识别对系统资产的每一项威胁。

5. 攻击评估。需要把每项威胁分解为可能对系统的各个攻击，并给出这些攻击会以什么方式进行。可以使用攻击树（Schneier 1999）来分析可能的攻击。这与故障树相类似，应该在树根的部分以一个威胁开始，并识别可能的攻击以及这些攻击将是如何进行的。

6. 控制识别。提出为保护资产而可能使用的控制方式。这些控制是通过一些技术机制来保护资产，比如加密。

7. 可行性评估。评估技术的可行性以及所提出的控制的开销。使用昂贵开销的控制来保护价值不高的资产是不值得的。

8. 信息安全需求定义。使用关于暴露、威胁和控制评估的知识得出系统信息安全需求。它们可能是系统基础设施的需求或是应用系统的需求。

Mentcare 病人管理系统是一个信息安全关键的系统。图 13-6 和图 13-7 是记录软件系统风险分析的报告片段。图 13-6 是资产分析，它描述了系统中的资产及其价值。图 13-7 显示了系统可能面临的一些威胁。

资 产	价 值	暴 露
信息系统	高。要求支持所有临床咨询。潜在安全关键	高。财物损失，因为诊所可能会被注销营业执照。恢复系统的成本。患者可能受到伤害，如果治疗不按规定的话
病人数据库	高。要求支持所有临床咨询。潜在安全关键	高。财物损失，因为诊所可能会被注销营业执照。恢复系统的成本。患者可能受到伤害，如果治疗不按规定的话
单条病人记录	通常低，虽然对特定的身份显赫的病人可能是高	直接损失低但可能影响声誉

图 13-6 Mentcare 系统初步风险评估报告中的资产分析

一旦初步的风险评估完成，就可以提出需求，对系统来说旨在规避、检测和缓解风险。然而，创建这些需求不是一个刻板或自动化的过程，需要工程师和领域专家基于他们对风险分析和软件系统功能性需求的理解来提出建议。下面是 Mentcare 系统信息安全需求和相关风险的一些例子：

1. 在诊疗阶段的开始，病人信息必须被下载，从数据库下载到系统客户端的一个安全区域。

威 胁	可能性	控 制	可行性
未经授权的用户以系统管理员的身份获取访问权限，使得系统不可用	低	只允许从物理上安全的地方进行系统管理	实现成本低，但是必须小心密钥的分发，并确保在紧急情况下能得到密钥
未经授权的用户以系统用户的身份访问系统，获取机密信息	高	要求所有用户使用生物机制进行身份验证。记录所有对病人信息的更改以跟踪系统的使用情况	技术可行但解决方案成本高。可能用户会抵触 简单并且对实现透明，也支持恢复

图 13-7 初步风险评估报告中的威胁和控制分析

风险：来自拒绝服务攻击的破坏。维护本地副本意味着仍然可以访问。

2. 系统客户端中的所有病人信息应该被加密。

风险：病人记录的外部访问。如果数据被加密，那么攻击者必须有加密密钥来发现病人信息。

3. 当诊疗阶段结束时，应该把所有的病人信息都上传到数据库并删除系统客户端的备份。

风险：通过盗取笔记本电脑来从外部访问病人记录。

4. 对系统数据库的所有修改以及这些修改的创建者信息应该生成日志并保留在不同于数据库服务器的另一台计算机上。

风险：内部或外部攻击破坏当前数据。日志应该允许从备份中重新创建最新记录。

前两条需求是相关的——病人的信息被下载到本地机器，这样即使病人数据库服务器受到攻击或者变得不可用，会诊也可以继续。然而，必须删除这些信息，这样才能避免后面的客户端电脑使用者访问这些数据。第四个需求是一个恢复和审计需求。它意味着所做出的修改可以通过回放修改日志进行恢复并且可以发现是谁进行的修改。这个机制可以阻止授权员工对系统的误用。

13.3.1 滥用案例

从风险分析得到信息安全需求是工程师和领域专家的创造性过程。已经为 UML 用户开发了一种方法来支持这个推导过程，这种方法采用的是滥用案例的想法（Sindre and Opdahl 2005）。滥用案例是表示与系统进行恶意交互的场景。我们可以使用这些脚本来讨论和明确可能的威胁，从而确定系统的信息安全需求。当要分析系统需求（第 4 和 5 章）时，这些脚本可以和用况结合在一起使用。

滥用案例与用况实例相关，并代表与这些用况相关的威胁和攻击。它们可以包含在用况图中，但应该有个更完整和详细的文本描述。图 13-8 中是使用 Mentcare 系统的医疗接待员的用况图，并增加了滥用案例。滥用案例通常用黑色椭圆表示。

与用况一样，滥用案例可以用几种方法来描述。我认为将它们描述为原始用况描述的补充是非常有帮助的。而且，滥用案例最好有灵活的格式，因为不同类型的攻击必须以不同的方式来描述。图 13-9 显示了传输数据用况（图 5-4）的原始描述，并添加了滥用案例描述。

滥用案例的问题反映了用况的通用问题，即最终用户和系统之间的交互不能捕获所有的系统需求。

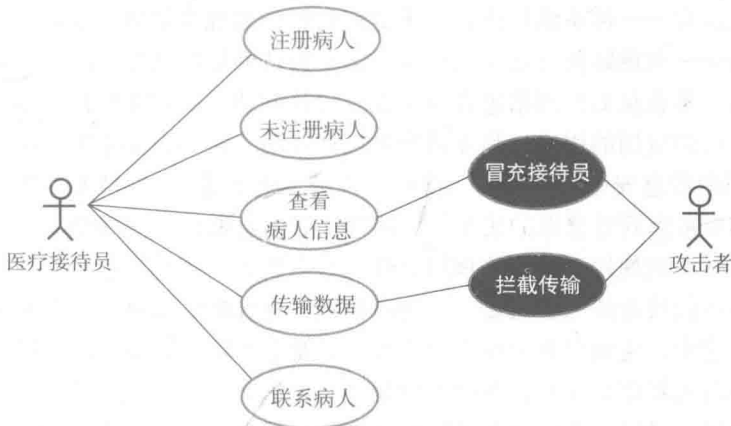


图 13-8 滥用案例

Mentcare 系统：传输数据	
参与者	医疗接待员、病人记录系统（PRS）
描述	医疗接待员可以从 Mentcare 系统向健康管理机构维护的通用的病人记录数据库传输数据。所传输的信息可以是更新的个人信息（地址、电话号码等）或者病人的诊断和治疗情况总结
数据	病人的个人信息、治疗总结
触发激励	医疗接待员发出的用户命令
响应	PRS 已经更新的确认信息
注解	接待员必须具有访问病人信息和 PRS 的适当的信息安全许可
Mentcare 系统：拦截传输（滥用案例）	
参与者	医疗接待员、病人记录系统（PRS）、攻击者
描述	医疗接待员把数据从他的电脑传输到服务器上的 Mentcare 系统。攻击者截获数据并获取该数据的副本
数据（资产）	病人的个人信息、治疗总结
攻击	将网络监控器添加到系统中，从医疗接待员到服务器的数据包被截获。在医疗接待员和数据库服务器之间设置一个假服务器，以便医疗接待员相信他们是与真的系统交互
缓解措施	所有网络设备必须锁在房间里。工程师访问设备必须经过认证。所有客户端与服务器之间的数据传输必须加密。必须使用基于证书的客户 - 服务器通信
需求	所有客户端与服务器的通信必须使用安全套接字层（SSL）。HTTPS 协议使用基于证书的认证和加密

图 13-9 误用案例描述

滥用案例可以作为信息安全需求工程过程的一部分，但你还需要考虑与系统利益相关者相关的风险，虽然他们不与系统直接交互。

13.4 信息安全系统设计

在系统部署后，再想将信息安全性措施加入系统是非常困难的。因此，需要在系统设计过程中就将信息安全问题考虑在内，并为增强系统的信息安全性而做出设计决策。本节侧重于讨论如下两个独立于应用的安全系统设计问题：

1. 体系结构设计——体系结构设计决策是如何影响系统的信息安全的?

2. 好的实践——在设计信息安全系统时什么是公认的好的实践?

当然, 这些还不是仅有的对信息安全重要的设计问题。每一个应用都是不同的, 信息安全设计也必须考虑到应用的用途、临界状态和操作环境。例如, 如果开发一个军事系统, 我们需要采用他们的信息安全分类模型(秘密、机密、绝密等)。如果要设计一个存储个人信息的系统, 我们要考虑到数据保护法案, 该法案针对信息如何管理规定了约束。

冗余性和多样性的使用, 对于实现可靠性是很重要的, 意味着系统能够抵抗那些针对特定设计和实现特性的攻击并从中恢复。支持高级别可用性的机制可以帮助系统从所谓的拒绝服务攻击中恢复过来, 这些攻击中攻击者的目标是使系统陷入瘫痪并让它停止正常工作。

设计一个系统要保证其安全, 不可避免地要有所取舍。为系统设计融入多种信息安全方案是完全有可能的, 这样会降低攻击成功的可能性。然而, 信息安全方案通常需要大量的额外计算并且影响系统的整体性能。例如, 我们可以通过加密机密信息来降低其被轻易破解的可能性, 但是, 这就意味着信息的用户也必须等待信息解码, 这可能会使得他们的工作变慢。

信息安全和系统可用性之间也有制约关系。信息安全方案有时要求用户记住并提供附加信息(例如, 大量的密码)。但是, 有时用户会忘记这些信息, 所以附加的信息安全意味着他们不能方便地使用系统。

因此设计者必须在信息安全、系统性能和可用性之间找到一个平衡点。这将取决于系统的类型和它会被用在什么地方。例如, 在一个军事系统中, 用户会对高级别信息安全系统很熟悉, 因此愿意接受并执行频繁的检查流程。但是, 在一个股票交易系统中, 因为信息安全检查而中断操作将会是完全不可接受的。



拒绝服务攻击

拒绝服务攻击试图通过大量的服务请求去轰击(bombarding)一个网络系统, 以此来造成这个网络系统的瘫痪。这样的系统负载因为没有在设计之初考虑, 所以它们会将合法的系统请求排除在外。因此, 系统会因为过重的负载而崩溃, 或者系统管理员将其脱机以停止大量请求的涌入, 这样都会导致系统变得不可用。

<http://software-engineering-book.com/web/denial-of-service>

13.4.1 设计风险评估

需求工程中的信息安全风险评估能够识别系统的一系列高级信息安全需求。然而, 随着系统的设计和实现, 在系统设计过程中做出的体系结构和技术决策会影响系统的信息安全性。这些决策产生新的设计需求, 并且可能意味着现有需求必须改变。

系统设计和设计相关风险的评估是交错的过程(图 13-10)。做出初步设计决策, 并评估与这些决策相关的风险。这种评估可能导致新的需求, 以减轻已识别的风险或设计更改产生的风险。随着系统设计的演进和更加详细的开发, 风险被重新评估, 并把结果反馈给系统设计者。设计风险评估过程在设计完成并且剩余风险可以接受时结束。

在评估设计和实施过程中的风险时, 我们能得到更多需要保护的信息, 还将了解系统中

的漏洞。这些漏洞中的一部分是所做的设计选择中固有的。例如，基于密码的认证的一个固有漏洞是授权用户向未经授权的用户透露其密码。因此，如果使用基于密码的认证，则风险评估过程可以提出新的需求以减轻风险。例如，可能需要多因素认证，其中除了密码外，用户必须使用一些个人信息来认证自己。

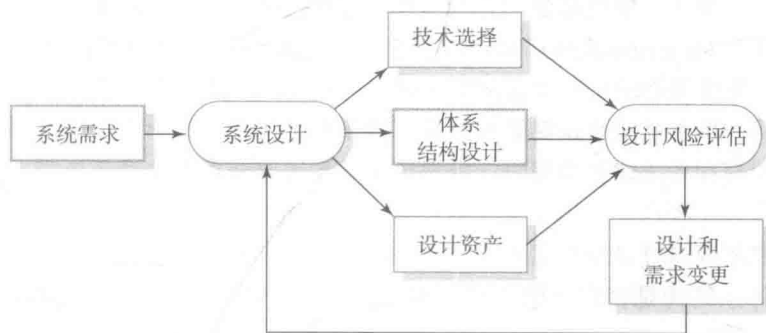


图 13-10 交错的设计和风险评估

图 13-11 是设计风险评估过程的模型。初步风险分析和设计风险评估之间的主要区别在于，在设计阶段，我们拥有关于信息表示和分发的信息以及需要保护的高级资产的数据库组织。我们还了解重要的设计决策，如要复用的软件、基础设施控制和保护等。根据这些信息，我们的评估可以识别对安全需求和系统设计的更改，为重要的系统资产提供额外的保护。

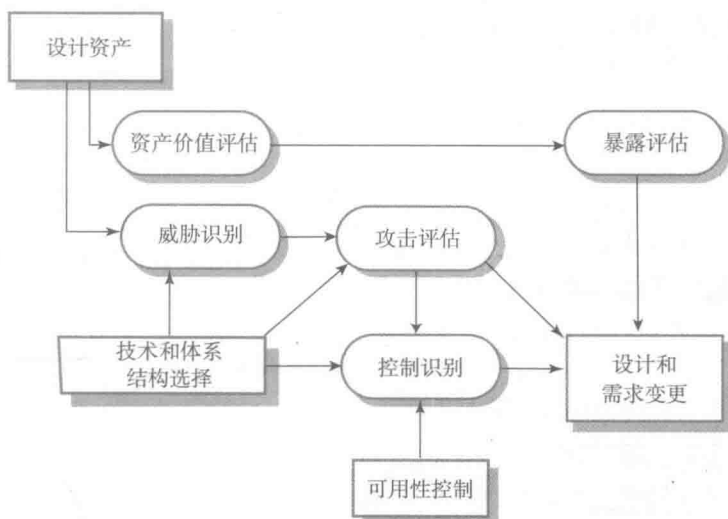


图 13-11 设计风险评估

来自 Mentcare 系统的两个例子说明了保护需求如何受到关于信息表示和分发的决策的影响：

1. 我们可能会做出一个设计决策来将患者个人信息和其接受治疗的信息分开，通过一个键值（key）链接这些记录。治疗信息远不如患者个人信息敏感，所以不需要大量保护。如果键值被保护，那么攻击者就只能获得常规信息，而不能链接到患者个人信息。

2. 假设从本节开始，设计决策是将患者记录拷贝到本地客户端系统。在服务器出现故障

时也将允许工作继续进行。这让卫生保健的工作者即使在没有网络连接的情况下也能够从笔记本电脑中获取患者记录。但是，我们现在有了两套要保护的记录和客户端副本，它们容易面临额外的风险，比如，盗取笔记本电脑。因此，我们必须思考应该用什么样的控制来降低风险。我们可能需要对保存在笔记本电脑或其他个人计算机上的客户端记录进行加密。

为了说明开发技术上的决定如何影响信息安全性，我们假设医疗部门已经决定了用商业现成信息系统建立 Mentcare 系统（他们自己的心理健康治疗病人信息系统）来保存病人的记录。此系统需要为使用它的每一类诊所做不同的配置。此决定已经做出，因为它会给绝大多数普遍使用的功能带来最低的开发成本和最快的开发时间。

当基于某个现存系统建立应用时，就必须接受系统原开发者的设计决策。我们假设一些设计决策如下：

- 1. 系统用户是通过登录用户名 / 口令组合认证的。不支持其他认证方法。
- 2. 系统的体系结构是客户 - 服务器，客户通过客户端计算机上的标准 Web 浏览器访问数据。
- 3. 信息以可编辑 Web 表单形式展现在用户面前。用户可以适当地改变信息并上传给服务器。

对于通用系统，这些设计决策是完全可以接受的，但是设计风险评估表明它们都有相关的漏洞。一些可能的漏洞例子如图 13-12 所示。

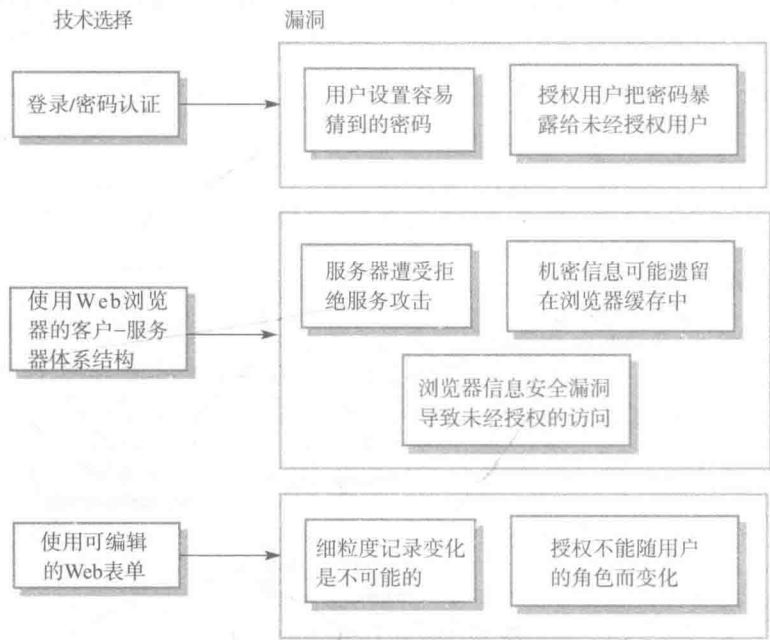


图 13-12 和技术选择相关的漏洞

漏洞一旦被发现，我们就必须决定采取哪些措施来降低相应的风险。这一般包括是否添加额外的系统安全需求或者改变使用系统的操作过程。这里无法讨论所有的有关固有漏洞的需求，下面列出其中一些需求：

- 1. 口令检查程序应该是可用的且应该每天运行。登记在系统字典中的用户口令会被识别并将弱口令的用户报告给系统管理员。

2. 对系统的访问只能是在系统管理员所认可和注册的客户计算机上进行。

3. 所有的客户计算机都只能安装一个由系统管理员所认可的 Web 浏览器。

在使用商业现成系统时,系统自身不可能包含口令检查程序,所以必须使用一个独立的系统。口令检查程序在用户口令被设定时分析其强度,如果用户选择了一个强度很弱的口令,那么它会通知用户。因此,易受攻击的口令能被很快发现并采取措施让用户更改其口令。

第二个和第三个需求意味着用户将总是从同一个浏览器上访问系统。我们可以在系统部署的时候决定哪个浏览器是最安全的,并且将它安装到所有客户端计算机上。信息安全更新得到简化,因为没有必要在发现和修补信息安全漏洞的时候更新不同的浏览器。

图 13-10 中所示的过程模型假定了一个设计过程,其中设计在开始实施之前已达到相当详细的级别。这不是敏捷过程的例子,敏捷过程的设计和实现是并行的,并且在代码重构时不断改进设计。频繁交付系统增量在时间上并不允许进行详细的风险评估,即使有关资产和技术选择的信息是触手可及的。

围绕安全和敏捷开发的问题已经被广泛讨论(Lane 2010;Schoenfield 2013)。到目前为止,这个问题还没有真正得到解决——有些人认为信息安全和敏捷开发之间存在根本性冲突,而其他一些人认为这种冲突可以使用以安全为重点的故事来解决(Safecode 2012)。这对于敏捷方法的开发者来说仍然是一个突出的问题。同时,许多拥有安全意识的公司拒绝使用敏捷方法,因为这些方法与信息安全和风险分析策略相冲突。

13.4.2 体系结构设计

软件体系结构的选择对系统特性有深刻的影响。如果使用不恰当的体系结构,维护系统信息的信息安全性和完整性或是保证系统有一定的实用性就是不可行的了。

在设计能维护信息安全性的系统体系结构时,需要考虑两个基本问题:

1. 保护——如何组织系统使其关键资产能在外部攻击时得到保护?
2. 分布——如何对系统资产进行分布使得逞的攻击数减到最少?

这些问题总是存在冲突的。如果将所有的资产放在一个地方,那么就可以在其上建立多层保护。因为只要建立一个保护系统,就能够为一个强大的系统提供多层保护。然而,如果保护失败,那么所有的资产就要受到损害了。增加若干个保护层也会影响系统的可用性,所以这就意味着要达到系统可用性和性能的要求更加困难。

另一方面,如果将资产分开存放,保护的成本会更高,因为保护系统在每个备份中都要实现。我们往往没有办法负担起这么多保护层的开支。保护伞被突破的概率也会大很多。但是,如果真的发生攻击,却不至于损失殆尽。将信息资产备份和分开是有可能的,如果一个备份被毁坏或是不可存取,那么其他的备份还可以使用。但是,如果信息是保密的,保留额外的副本会增加入侵者获得这个信息的风险。

对于病人记录系统而言,我们使用具有共享中央数据库的客户-服务器体系结构。为了对系统提供保护,系统使用分层的体系结构,使关键性的受保护资产位于系统的最底层,并在其上设置各种不同的保护。图 13-13 是有关多层系统体系结构的说明,被保护的关键性资产是每个病人的记录。

为了访问和修改病人的记录,攻击者需要穿越 3 个系统层:

1. 平台层保护。最顶层控制对病人记录系统所在的平台的访问、这通常涉及用户到某个

特别计算机的注册。平台也一般包括对系统上文件完整性维护的支持、备份等。

2. 应用层保护。下一个保护层是建立在应用本身上的。它包括用户对应用的访问、用户身份认证以及对执行像浏览或修改数据等动作的许可。应用专门的完整性管理支持是可利用的。

3. 记录层保护。当需要访问特殊记录且在此记录上执行所请求的操作时，对用户的授权检查调用这一层。此层的保护也可能包括加密来确保记录不会被文件浏览器所浏览。完整性检查，例如使用密码校验和，能检测出在正常记录更新机制之外产生的改变。



图 13-13 分层保护体系结构

对任何应用来说，保护层数量依赖于数据的危险程度。不是所有的应用都需要记录层保护，因此，通常使用更大粒度的访问控制。为达到信息安全性，我们不能在每一层上使用相同的用户许可。理论上，如果是一个基于口令的系统，那么应用口令应该不同于系统口令，也不同于记录层口令。但是，大量的口令对于用户记忆来说是困难的，并且他们会被重复的认证请求所激怒。因此，我们经常不得不考虑系统可用性而在信息安全方面有所取舍。

如果对数据的保护是一个关键性需求，那么集中的客户-服务器体系结构通常是最有效的信息安全体系结构。服务器负责保护敏感数据。然而，如果保护失败，则攻击所带来的损失可能很高，所有的数据都有可能丢失或受损，恢复的成本也将很高（例如，所有用户的许可都需要重新颁发）。集中的系统在拒绝服务攻击方面是脆弱的，拒绝服务攻击加重了服务器的负载，使之不能让任何用户访问到系统数据库。

如果服务器漏洞的后果很严重，我们可以决定让应用采用分布式体系结构。在此情形下，系统资产被分布到多个不同的平台上，每个平台有自己的保护机制。对某个节点的攻击可能造成某些资产不可用，但是还会有别的系统服务仍然保持可用。数据可以在系统的不同节点中复制，这样从攻击中恢复就简单了。

图 13-14 给出的是在纽约、伦敦、法兰克福以及香港市场上使用的银行股票和基金交易系统的体系结构。它是一个分布式结构，每个市场的数据都是单独维护的。用于支持要求极高的股票交易（用户账户和价格）的关键活动所需的资产已经复制到各个节点上。如果系统

的某个节点受到攻击变得不可用，股票交易的关键活动能够转移到另一个国家或地区，这样对用户来说就仍然是可用的。

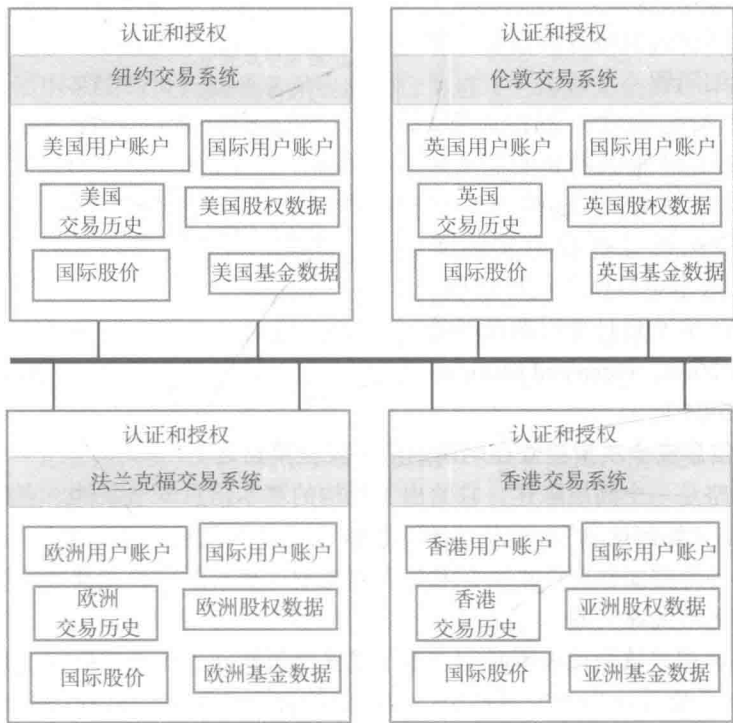


图 13-14 股权交易系统中的分布式资产

前面已经讨论过在信息安全和系统性能之间寻找平衡点的问题。很多时候，信息安全系统设计的一个问题是，能提供安全保障的最合适的体系结构可能不能达到最好的性能要求。例如，假设一个应用的根本需求是维护某个大型数据库的机密性，对它的另一个需求是能快速访问数据。高级别保护建议提供多个保护层，这就意味着系统的层次之间必须要通信。这就不可避免地产生性能开销，从而降低数据访问的速度。

如果采用另外的体系结构类型，那么实现保护和保证机密性会更加困难且需要更高的代价。在此情况下，你必须与购买系统的客户讨论内在的冲突，在如何解决问题的方案上达成一致意见。

13.4.3 设计准则

并不存在一个如何达到系统信息安全性的硬性规定。不同类型的系统需要不同的技术措施来达到系统拥有者所能接受的安全等级。不同用户组的态度和需求对什么是可以接受的以及什么是不能接受的有深深的影响。例如，对于银行来说，用户可能接受较高等级的安全性，因此比高校系统有更多的安全性程序。

然而，还是有一些一般性准则，它们广泛适用于系统信息安全性的方案设计，概括了信息安全系统的一些好的设计实践。信息安全性的一般性设计准则有两个主要的用途：

1. 它们能够提高软件工程团队对信息安全问题的重视。软件工程师通常只关注近期目标，如怎样让软件能用和能尽快交付给客户。信息安全问题很容易被忽视。这些准则能使信

息安全问题在软件设计决策制定之时就得到考虑。

2. 它们能够用在系统有效性验证过程的评审清单中。从这里讨论的高层准则，可以导出更多探讨如何将信息安全设计到系统中的详细问题。

安全准则有时是非常一般的原则，例如“保护系统中最脆弱的链接”“保持简单”和“避免模糊”。这些一般准则太模糊，无法在设计过程中实际使用。因此，这里给出更具体的设计准则。图 13-15 中总结的 10 个设计准则来自不同的来源 (Schneier 2000; Viega and McGraw 2001; Wheeler 2004)。

信息安全系统工程的设计准则

1. 将信息安全决策建立在明确的信息安全策略之上
2. 避免单点失效
3. 可恢复性失效
4. 寻求信息安全和可用性间的均衡
5. 记录用户行为
6. 通过冗余性和多样性降低风险
7. 验证所有输入
8. 划分资产
9. 部署设计
10. 可恢复性设计

图 13-15 信息安全系统工程的设计准则

准则 1：将信息安全决策建立在明确的信息安全策略之上

信息安全策略是一个高层陈述，它给出了机构的基本信息安全条件。它要定义信息安全“是什么”而不是“如何达到”。此策略不应该规定提供和执行信息安全的机制。一般来讲，信息安全策略的各个方面都应该反映在系统需求当中。在实践中，尤其是使用敏捷开发的时候，做到这一点是不太现实的。

设计者在制定和评估设计决策时应该把信息安全策略作为框架。假如你在为本章前面提到过的 Mentcare 系统设计访问控制系统。医院的信息安全策略是只有认可的临床工作人员可以修改病人的电子记录。因此你的系统必须包括检验每一个试图修改信息的人的合法性的机制以及拒绝无资格人员非法修改的机制。

你可能遇到的问题是很多机构没有明确的系统信息安全策略。随着时间的推移，系统可能已经为了解决发现的问题做了更改，但是并没有详细的策略文档来指导系统进化。在这样的情况下，你需要从例子中制定出策略并写成文档，然后让公司管理者对它进行确认。

准则 2：避免单点失效

在任何要求极高的系统中，努力做到避免单点失效是个好的设计实践。这意味着系统某部分的单个失效不会造成系统的总体失效。用信息安全的术语，这意味着我们不应该依赖单一机制来确保信息安全，而是应该使用多个不同技术。有时我们称之为“深度防御”。

深度防御的一个典型例子是多重身份认证。例如使用口令来鉴别系统的用户，也要包括挑战/响应验证机制使得用户必须预先在系统中注册问题和答案。经过口令鉴别，用户必须正确回答问题方可进入系统。

准则 3：可恢复性失效

在所有系统中系统失效都是不可避免的，如同可恢复性失效之于安全要求极高的系统，信息安全要求极高的系统也必须总是“失效可恢复的”。在系统失效时可使用的后退方式不能有比系统本身更差的安全性。系统失效也不能让攻击者访问到一般情况下不可能访问到的数据。

例如，在病人信息系统中，作者建议了一个需求，即病人的数据应该在病人一进入诊所就下载到系统客户端。这样做能加速访问且能在服务器无法连接的时候仍可以访问到。通常，当病人看完病时服务器就删除此部分数据。然而，如果服务器失效，就存在信息被保留

在客户端的可能。在这些情况下的可恢复性失效方法可以是对客户端病人数据进行加密。这就意味着未经授权的人不会读到这些数据。

准则 4：寻求信息安全和可用性间的均衡

信息安全要求和可用性要求有时会发生冲突。为了让系统安全，必须引入多个检查，包括确保用户获准使用系统的检查以及确保他们的行为遵守了信息安全策略。所有这些不可避免地要对用户施加要求——他们可能需要记住登录名字和口令，只能从某一台计算机上使用系统等。这些意味着用户在启动系统和有效的使用上需要更多的时间。当你添加这些信息安全特性到系统中的时候，不可避免地要降低系统的可用性。作者推荐过 Cranor 和 Garfinkel 的书 (Cranor and Garfinkel 2005)，其中对一般领域中的信息安全和可用性的广泛问题进行了讨论。

添加新的信息安全特性到系统中是要付出可用性代价的，这里存在一个平衡点，越过这个平衡点再增加信息安全特性就会达不到预期的目标。例如，如果你要求用户输入多个口令或者是需要频繁地更换口令，势必造成用户无法记住口令，他们可能就自然地会将口令抄在本子上。攻击者（尤其是内部人员）就很容易发现此口令并由此访问系统。

准则 5：记录用户行为

如果可以，我们应该始终保存一个用户行为日志。这个日志至少应该记录谁做了什么、使用的资产，以及行为的时间和日期。如准则 2 中所提到的，如果我们把用户行为保存为一个可执行命令表，那么就可以在失效后重新执行记录来恢复系统。当然，我们也需要工具来分析记录并且检测可能异常的行为。这些工具能够扫描记录并找出异常行为，因此可以帮助检测攻击和跟踪攻击者是如何获得系统使用权的。

除了帮助系统从失效中恢复以外，用户行为日志作为一个威慑，对内部攻击行为也起到了很大作用。如果人们知道他们的行为会被记录，那么他们就不太可能做一些未经授权的事情。这对于偶然性攻击是有很有效的（如一名护士查找病人记录），对于检查通过社会工程盗取合法用户证书的攻击也很有效。当然，这并不十分简单，技术上相当熟练的内部人员也能够获得并修改记录。

准则 6：通过冗余性和多样性降低风险

冗余性是指需要在系统中维护多个版本的软件和数据。对于软件来说，多样性意味着不同版本不应该基于相同的平台或者是使用相同的技术。因此，平台或技术漏洞将不会影响所有版本，也就无法带来同样的失效。

我们已经讨论过冗余性的两个例子——第一个是 Mentcare 系统中在服务器和客户端维护病人信息，第二个是如图 13-14 所示的分布式股票交易系统。在病人记录系统中，可以在客户端和服务端使用不同操作系统（例如服务器使用 Linux，客户端使用 Windows），这样确保基于操作系统漏洞的攻击无法同时影响服务器和客户端。当然，我们必须接受在一个机构中维护不同操作系统所增加的管理开支作为获得这些好处的代价。

准则 7：验证所有输入

对系统的常见攻击总是通过一些未预料到的输入让系统以一种无法预知的工作方式工作。这些会直接造成系统崩溃，导致无法服务或者是输入会混合进恶意代码。缓冲区溢出漏洞最早出现在互联网蠕虫 (Spafford 1989) 中且常常被攻击者采用，它就是使用长输入串触发的。所谓的“SQL 中毒”，即恶意用户输入一个 SQL 片段让服务器解释，是另一个相当普遍的攻击。

如果我们定义系统输入的格式和结构,可以避免许多这样的问题。这种定义应该基于你关于系统输入的知识。例如,如果要输入姓,就应该保证是字母,没有数字或标点符号(唯一可用的标点符号是连字号)。同时还要拒绝明显过长的输入。例如,没有谁的姓超过40个字符,也没有一个住址会多于100个字符。如果输入是数字,就不应该有字母。系统实现时,输入检查中应该有这些信息。

准则 8: 划分资产

分割意味着不用提供对系统中所有信息的访问权限。基于一般的“需要了解”的信息安全原则,你应当将系统中的信息合理地划分到一些仓位,从而使得用户只能访问到他工作所需要的信息而不是全部系统信息。这意味着攻击的影响被局部化。有些信息会丢失或者被损坏,但是不太可能造成对系统中所有信息的全面影响。

例如,在病人信息系统中,所做的设计应该是这样的:在任何一个诊所中,诊所工作人员一般只能访问到预约此诊所的病人的记录。他们一般不应该看到系统中所有病人的信息。这不仅有利于限制由于可能的内部攻击造成损失,也意味着即使入侵者窃取了他们的证书,也不能破坏所有的病人记录。

说到此,我们还应该让系统有这样一种允许未预料访问的机制,即假设有病情十分严重的病人需要紧急处理而无须事先预约。在此情况下,我们应该使用另外的安全机制来阻止系统中的信息分割。在这样的情况下,信息安全会被放宽来维持系统的可用性,使用日志机制来记录系统使用是必要的,然后我们能检查日志来跟踪任何未经授权的使用。

准则 9: 部署设计

很多信息安全问题的发生是因为将系统部署在运行环境中时没有进行正确的配置。部署意味着在要执行软件的计算机上安装软件,并设置软件参数以反映系统用户的执行环境和首选项。忘记关闭调试设备或忘记更改默认管理密码等错误可能会在系统中引入漏洞。

好的管理实践会避免由于配置和部署错误而产生的很多信息安全问题。然而,软件设计者有责任“为部署而设计”。我们应该总是提供对部署的支持,以降低系统管理者(或者用户)在配置软件过程中犯错误的概率。

作者推荐4种能在系统中加入部署支持的方法:

1. 支持查看和分析配置。应该始终在系统中包括允许管理员或经过授权的用户检查系统的当前配置的功能。
2. 最小化默认权限。所设计的软件应该让系统的默认配置提供最低的基本权限。
3. 本地化配置设置。在设计系统配置支持时,应确保配置中影响系统相同部分的所有内容都设置在同一位置。
4. 提供简单的方法来修复信息安全漏洞。应该包括用于更新系统以修复发现的信息安全漏洞的简单机制。

部署问题不像以前那样是问题了,因为越来越多的软件不需要客户端安装。相反,软件作为服务运行并且通过Web浏览器访问。然而,服务器软件仍然易于部署错误和遗漏,并且一些类型的系统需要在用户的计算机上运行专用软件。

准则 10: 可恢复性设计

无论在系统信息安全维护上花多大的力气,都应该总是假设信息安全失效是要发生的。因此,我们应该考虑如何从可能的失效中恢复,并重建系统使之处于安全的运行状态。例如,你会包含一个备份的验证系统以备你的口令认证被攻破。

假设某个未经批准的人从医院外部获得了对病人记录系统的访问权限，且我们不知道他如何获得有效登录名和口令的。我们需要重新初始化验证系统，并且不仅仅是修改被入侵者使用过的证书。这很有必要，因为入侵者可能也获得了其他用户的口令。因此，我们也必须改变所有已授权用户的口令信息，以确保未经授权的用户没有办法进入口令变更机制中来。

因此，需要设计系统使其拒绝对没有更改口令的所有用户的访问，直到他们完成了口令的修改为止，并且给所有的用户发邮件要求他们做出修改。我们需要一个替代机制去认证真正用户以允许他们更改口令，尽管假设他们选择的口令可能是不安全的。一个方法是使用挑战/响应机制，即用户必须回答他们事先注册的问题。这只有在修改口令的情况下被调用，使得系统从攻击中恢复过来，只有相对少量的用户受到影响。

可恢复性的设计是建立系统灵活性的基本要素。第 14 章中将更详细地介绍这个主题。

13.4.4 信息安全系统编程

信息安全系统设计意味着在应用系统中设计信息安全性。然而，除了关注设计层的信息安全性，在对软件系统编程时考虑信息安全性也很重要。许多对软件的成功攻击依赖于程序开发时引入的程序漏洞。

第一个广为人知的对基于互联网的系统的攻击发生在 1988 年，当一个蠕虫被引入网络中的 Unix 系统时 (Spafford 1989)。这利用了一个众所周知的编程漏洞。如果系统用 C 编程，则不进行自动数组边界检查。攻击者可以包括具有程序命令作为输入的长字符串，这会覆盖程序栈并且可以使控制转移到恶意代码。自那时以来，用 C 或 C++ 编程的许多其他系统已经被利用了此漏洞。

本示例说明了信息安全系统编程的两个重要方面：

1. 漏洞通常是语言特定的。数组边界检查在诸如 Java 之类的语言中是自动的，因此这不是可以在 Java 程序中利用的漏洞。然而，数以百万计的程序是用 C 和 C++ 编写的，因为它们可以开发更高效的软件。因此简单地避免使用这些语言不是一个现实的选择。

2. 信息安全漏洞与程序可靠性密切相关。上述示例导致相关程序崩溃，因此为提高程序可靠性而采取的措施也可以提高系统信息安全性。

在第 11 章中，介绍了可靠的系统编程的编程准则，如图 13-16 所示。这些指南还有助于提高程序的信息安全性，因为攻击者专注于程序漏洞以获取系统访问权限。例如，SQL 中毒攻击基于攻击者使用 SQL 命令填充表单，而不是系统预期的文本。这些可能会损坏数据库或释放机密信息。如果根据输入的预期格式和结构实施输入检查（指南 2），则可以完全避免此问题。

可靠的编程准则

1. 限制程序中信息的可见性
2. 检查所有输入的有效性
3. 为所有异常提供处理程序
4. 最小化易出错结构的使用
5. 提供重新启动功能
6. 检查数组边界
7. 调用外部构件时包含超时
8. 命名表示真实世界值的所有常量

图 13-16 可靠的编程准则

13.5 信息安全测试和保证

对系统信息安全性的评估越来越重要，这使得我们可以确信我们使用的系统是安全的。因此，基于 Web 的系统的验证和确认过程应集中于信息安全评估，其中测试系统抵抗不同类型攻击的能力。但是，正如 Anderson (Anderson 2008) 所说，这种信息安全评估很难开展。结果，系统总存在可使攻击者获得存取权、摧毁系统或破坏数据的安全漏洞。

从根本上讲，信息安全很难评估的原因有两个：

1. 就像某些安全需求一样，信息安全需求是“不应该”的需求。这就是说，它们定义了哪些系统行为是不允许发生的，而不是定义了期待发生的行为。然而，并不总是能用一些系统容易检查的简单约束来定义这些行为。

如果资源可用（至少在原理上），你总能证明系统满足功能性需求。但是很难证明系统不能做某事。所以系统不管有多少的测试，信息安全漏洞仍保留在系统中。

当然，你可以生成一些功能性的需求，设计它们来保护系统免遭一些已知类型的攻击。但是，你不能由未知的和不能预期类型的攻击派生出相应的需求。即使对一个久经考验的系统，机灵的攻击者也能发现新的攻击方式，从而可以进入看似安全的系统中。

2. 攻击系统的人都很聪明，并且很积极地去发现系统的漏洞。他们很乐意拿系统做实验，并且尝试那些非同寻常的行为和系统使用。例如，在一个姓氏域内，他们可能输入1000个混合着字母、标点和数字的字符来看看系统怎么反应。

此外，一旦他们发现一个漏洞，他们就会交换关于漏洞的相关信息来增加可能攻击者的数目。他们已经建立了互联网论坛，以交换有关系统漏洞的信息。在恶意软件领域还有一个蓬勃发展的市场，其中有攻击者可以访问的工具包，帮助他们轻松地开发恶意软件，如蠕虫和击键记录器。

攻击者可能尝试发现系统开发者所做出的假设，然后做出违反这些假设的动作以观察出现怎样的结果。他们处在一个利用和探索系统的时期，并且使用软件工具来分析系统，以发现可以被他们利用的漏洞。实际上，他们可能比系统测试工程师花更多的时间来查找漏洞，作为测试者也必须将焦点集中在测试系统之上。

我们可以结合使用测试、基于工具的分析 and 形式化验证来检查和分析应用系统的信息安全性：

1. 基于经验的测试。在这种情形中，检验小组根据掌握的攻击类型对系统进行分析。这可能包括开发测试用例或者检查系统源代码。例如，检查系统是否容易遭受众所周知的SQL中毒攻击时，可以使用包含SQL语句的命令测试系统。检查缓冲区溢出错误是否会发生时，可以检查所有输入缓冲区，看是否程序中缓冲区元素的赋值都在边界内。

2. 渗透测试。这是一种基于经验的测试形式，可以从开发团队外部利用经验来测试应用系统。专门建立一个渗透测试小组，其使命就是攻破系统信息安全防线。他们模拟黑客对系统攻击，发挥他们的才智来发现系统漏洞。渗透测试小组成员应当具有原先信息安全测试工作的经验，并且找到过系统信息安全性的弱点。

3. 基于工具的测试。这种方法是使用各种不同的信息安全工具如口令检查器分析系统。口令检查器检查不安全的口令，如姓名或由连续字母组成的字符串。这实际上是基于经验的验证的扩展，系统信息安全缺陷的经验被固化在工具中了。当然，静态分析是另一种使用率日益提高的基于工具的测试方法。

基于工具的静态分析（第12章）是一种特别有用的信息安全检查方法。对程序的静态分析可以快速地将测试团队引导到程序中可能包含错误和漏洞的区域。在静态分析中显示的异常可以直接修复，或者可以帮助识别需要进行的测试以揭示这些异常实际上是否表示系统的风险。微软使用静态分析来检查其软件是否存在可能的信息安全漏洞（Jenney 2013）。惠普提供了一种名为Fortify（Hewlett-Packard 2012）的工具，专门用于检查Java程序的信息安全漏洞。

4. 形式化验证。我们在第10章和第12章讨论了形式化程序验证的使用。本质上,这涉及形式化的数学论证,证明程序符合其规格说明。Hall 和 Chapman (Hall and Chapman 2002) 十多年前指出证明系统满足其正式信息安全需求的可行性,并且自那时以来已经进行了许多其他实验。然而,与其他领域一样,对信息安全的形式化验证没有得到广泛使用。它需要专家的专业知识,不可能像静态分析那样具有合理的成本效益。

信息安全测试不可避免地受到测试团队可用的时间和资源的限制。这就意味着应当采用基于风险的方法来进行信息安全测试,并且把注意力集中在你认为系统最可能出现问题的风险上。如果你对系统的信息安全风险进行了分析,这些分析可以用来驱动测试的过程。在针对这些风险所导出的信息安全需求对系统进行测试之外,测试团队也应该试图通过采用另外的威胁系统资产的途径来攻击系统。

要点

- 信息安全工程专注于如何开发和维护能够抵御恶意攻击的软件系统,恶意攻击破坏的是基于计算机的系统或其数据。
- 信息安全威胁可能是对系统或其数据的机密性、完整性或可用性的威胁。
- 信息安全风险管理包括评估对系统的攻击可能造成的损失,并得出旨在消除或减少这些损失的信息安全需求。
- 为了定义信息安全需求,我们应该识别要保护的资产,并定义如何使用信息安全技术来保护这些资产。
- 设计信息安全系统体系结构时的关键问题包括组织系统结构以保护关键资产,分发系统资产以最小化成功攻击的损失。
- 信息安全设计准则使系统设计人员对他们可能未考虑到的信息安全问题提高敏感度,为创建安全审查清单提供了依据。
- 信息安全验证很困难,因为信息安全需求指出了系统中不应该发生什么,而不是应该发生什么。此外,系统攻击者是具有才智的,并且比信息安全测试者有更多的时间来探测弱点。

阅读推荐

《Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd ed.》对建立安全系统的问题进行了全面的讨论。重点在于系统而不是软件工程,广泛讨论了硬件和网络,以及来自实际系统失效的优秀示例。(R. Anderson, John Wiley & Sons, 2008) <http://www.cl.cam.ac.uk/~rja14/book.html>

《24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them》是有关安全系统编程的最好的实用书之一。作者讨论了最常见的编程漏洞,并描述了如何在实践中避免这些漏洞。(M. Howard, D. LeBlanc, J. Viega, McGraw-Hill, 2009)

《Computer Security: Principles and Practice》是关于计算机安全问题的一本好教材。它涵盖安全技术、可信系统、安全管理和加密。(W. Stallings, L. Brown, Addison-Wesley, 2012)

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap13/>

支持视频的链接：<http://software-engineering-book.com/videos/security-and-resilience/>

练习

- 13.1 解释应用信息安全工程和基础设施信息安全工程之间的重要区别。
- 13.2 对于 Mentcare 系统，除了本章中讨论的内容之外，请提供资产、暴露、漏洞、攻击、威胁和控制的例子。
- 13.3 解释为什么在系统开发期间需要进行初步信息安全风险评估和设计风险评估。
- 13.4 扩展图 13-7 中的表格，以确定 Mentcare 系统的另外两个威胁以及相关的控制。使用它们作为生成实现建议控制的软件信息安全需求的基础。
- 13.5 使用非软件工程环境中的类比解释为什么应该使用分层的资产保护方法。
- 13.6 解释为什么在开发安全系统时使用多种技术很重要。
- 13.7 对于 13.4.2 节讨论的股票交易系统，其体系结构如图 13-14 所示，提出对系统进一步的合理攻击，并提出对付这些攻击的可能策略。
- 13.8 解释为什么在编写安全系统时，要验证所有用户输入是否具有预期格式。
- 13.9 说明你会如何验证你开发的应用程序的密码保护系统。解释你认为可能有用的工具所具有的功能。
- 13.10 Mentcare 系统必须防范可能暴露病人个人信息的攻击。提出针对此系统的三种可能的攻击。使用此信息，扩展图 13-17 中的清单来帮助 Mentcare 系统的测试人员。

信息安全检查表
1. 在应用程序中创建的所有文件是否具有适当的访问权限？错误的访问权限可能导致这些文件被未经授权的用户访问。
2. 在不活动一段时间后，系统是否自动终止用户会话？处于活动状态的会话可能允许通过无人值守计算机进行未经授权的访问。
3. 如果系统是用编程语言编写的，没有数组边界检查，有没有缓冲区溢出可能被利用的情况？缓冲区溢出可能允许攻击者向系统发送代码字符串，然后执行它们。
4. 如果设置了密码，系统是否检查密码是否“强”？强密码由混合字母、数字和标点符号组成，不是正常的字典条目。它们比简单的密码更难破解。
5. 是否始终根据输入规范检查系统环境的输入？错误处理错误的输入是导致信息安全漏洞的常见原因。

图 13-17 信息安全检查表中的条目示例

参考文献

- Anderson, R. 2008. *Security Engineering*, 2nd ed. Chichester, UK: John Wiley & Sons.
- Cranor, L. and S. Garfinkel. 2005. *Designing Secure Systems That People Can Use*. Sebastopol, CA: O'Reilly Media Inc.
- Firesmith, D. G. 2003. "Engineering Security Requirements." *Journal of Object Technology* 2 (1): 53-68. http://www.jot.fm/issues/issue_2003_01/column6
- Hall, A., and R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18-25. doi:10.1109/52.976937.
- Hewlett-Packard. 2012. "Securing Your Enterprise Software: Hp Fortify Code Analyzer." <http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-2455ENW&cc=us&lc=en>

- Jenney, P. 2013. "Static Analysis Strategies: Success with Code Scanning." <http://msdn.microsoft.com/en-us/security/gg615593.aspx>
- Lane, A. 2010. "Agile Development and Security." <https://securosis.com/blog/agile-development-and-security>
- Pfleeger, C. P., and S. L. Pfleeger. 2007. *Security in Computing, 4th ed.* Boston: Addison-Wesley.
- Safecode. 2012. "Practical Security Stories and Security Tasks for Agile Development Environments." http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf
- Schneier, B. 1999. "Attack Trees." *Dr Dobbs Journal* 24 (12): 1–9. <https://www.schneier.com/paper-attacktrees-ddj-ft.html>
- . 2000. *Secrets and Lies: Digital Security in a Networked World.* New York: John Wiley & Sons.
- Schoenfeld, B. 2013. "Agile and Security: Enemies for Life?" <http://brookschoenfeld.com/?p=151>
- Sindre, G., and A. L. Opdahl. 2005. "Eliciting Security Requirements through Misuse Cases." *Requirements Engineering* 10 (1): 34–44. doi:10.1007/s00766-004-0194-4.
- Spafford, E. 1989. "The Internet Worm: Crisis and Aftermath." *Comm ACM* 32 (6): 678–687. doi:10.1145/63526.63527.
- Stallings, W., and L. Brown. 2012. *Computer Security: Principles, d Practice.* (2nd ed.) Boston: Addison-Wesley.
- Viega, J., and G. McGraw. 2001. *Building Secure Software.* Boston: Addison-Wesley.
- Wheeler, D. A. 2004. *Secure Programming for Linux and Unix.* Self-published. <http://www.dwheeler.com/secure-programs/>

韧性工程

目标

本章的目标是介绍韧性工程的思想，其含义是通过设计使得系统能够经受不利的外部事件（例如操作人员失误或网络攻击）的影响。阅读完本章后，你将：

- 理解韧性、可靠性、信息安全性的区别，以及理解韧性对于网络化系统为什么重要；
- 了解构造韧性系统的一些基础问题，即问题发现、失效和攻击防御、关键性服务恢复，以及系统复原；
- 理解韧性为什么是一个社会技术问题而不是一个纯技术问题，以及系统操作人员和管理人员在保证系统韧性过程中的角色；
- 了解一个可以实现韧性的系统设计方法。

1970 年 4 月，阿波罗 13 月球载人飞行任务遭遇了一次灾难性的事故（失效）。一个氧气瓶在太空中爆炸，导致空气中的氧气以及为飞船提供动力的燃料电池所用的氧气发生严重泄漏。这种情形威胁生命安全，无法救援。针对这种情形没有事故处理预案。然而，通过按照非常规的方式使用设备以及对标准规程的调整，飞船全体船员和地面工作人员一起努力解决了这个问题。飞船被安全带回地球，全体船员都得以幸存。这个系统（人、设备、过程）整体上是有韧性的，通过适应性调整成功应对了这次事故（失效）并从中恢复。

本书第 10 章中，作为系统可靠性的一个基本属性介绍了韧性的概念。其中对于韧性的定义如下：

系统的韧性是指系统在出现破坏性事件（例如设备失效和网络攻击）的情况下能够保持系统关键性服务的连续性的程度。

关于韧性并没有“标准”的定义。其他作者，例如 Laprie (Laprie 2008) 和 Hollnagel (Hollnagel 2006)，按照系统应对变化的能力给出了另一个泛化的定义，即一个有韧性的系统能够在系统设计者所做出的一些基本假设不再成立的情况下也能够保持正常运行。例如，一个系统最初的设计假设可能是用户会犯错但不会故意寻找系统的漏洞并加以利用。如果这个系统在一个容易遭受网络攻击的环境中使用，这个假设便不再成立。一个有韧性的系统可以应对这种环境变化并持续正常运行。虽然这些定义更加泛化，但是本书对于韧性的定义更加接近目前政府和工业界在实践中对这个术语的使用。其中包含了 3 个方面的基本思想：

1. 一个系统所提供的服务中有一些是关键性服务，这些服务的失效会造成严重的人员、社会或者经济损失。
2. 有些事件是破坏性的，可能会影响系统提供关键服务的能力。
3. 韧性是一种判断，没有关于韧性的度量标准，韧性也无法度量。一个系统的韧性只能由专家通过查看系统及其运行过程来评价。

关于系统韧性的基础性工作起源于安全关键系统社群，他们的目的是理解哪些因素有

助于避免事故以及在事故中生存。然而,针对网络化系统越来越多的网络攻击意味着韧性现在经常被视为一个信息安全问题。构造可以经受恶意网络攻击并保持用户服务的系统十分重要。

显然,韧性工程与可靠性和信息安全工程密切相关。可靠性工程的目的是保证系统不失效。系统失效是一个外部可观察的事件,往往是系统内部故障导致的结果。因此,如第11章中所讨论的故障避免和容错等技术相继被提出以减少系统故障数量,并在故障导致系统失效前进行控制。

即使我们已经竭尽全力,故障总是会在大型、复杂系统中出现并导致系统失效。交付时间很短,测试预算有限,开发团队在压力下工作,而在现实中发现一个软件系统中所有故障和信息安全漏洞几乎是不可能的。我们构造的系统是如此复杂(见第19章),以至于我们不可能理解系统构件之间的所有交互。这些交互中的一些可能触发系统的全局失效。

韧性工程并不关注避免失效,而是关注接受失效总会发生的现实。它有两个重要的假设。

1. 韧性工程假设系统失效无法避免,因此关注降低失效带来的损失以及如何从失效中恢复。

2. 韧性工程假设系统开发已经采用了好的可靠性工程实践来尽可能减少系统中技术故障的数量,因此更加强调减少由外部事件(例如,操作人员失误或网络攻击)导致的系统失效的数量。

实践中,技术系统失效经常是由系统外部事件引发的。这些事件可能包括意料之外的操作人员动作或者用户错误。然而,过去几年随着网络化系统数量的增长,这类事件经常与网络攻击相关。在网络攻击中,一个恶意的人或群体试图破坏系统或窃取机密信息。这些已经成为比用户或操作人员失误更加突出的系统失效来源。

由于我们假设失效发生不可避免,韧性工程既关注从失效中立即恢复以保证关键性服务的提供,又关注后续所有系统服务的恢复。正如14.3节中所讨论的,这意味着系统设计人员必须考虑能够保持系统的软件状态和数据的系统特性,从而保证当发生系统失效时重要的信息可以恢复。

系统问题的发现和恢复包括4个相关的韧性活动。

1. 发现。系统或其操作者应当能够发现可能导致系统失效的问题的症状。理想情况下,应当有可能在系统失效之前发现症状。

2. 防御。如果较早发现问题的症状或者网络攻击的迹象,那么可以调用防御策略来减少系统失效发生的可能性。这些防御策略可以关注将系统的关键性部分隔离开以使得它们不会受到其他地方问题的影响。防御包括主动防御(即系统中包含防御机制来应对问题)以及反应式防御(即在发现问题后再采取行动)。

3. 恢复。如果发生失效,那么恢复活动的目的是快速恢复关键性的系统服务以确保系统用户不会严重地受到失效的影响。

4. 复原。在这个最终的活动中,所有的系统服务都得到恢复,正常的系统操作可以继续。

这些活动中的系统状态变化如图14-1所示,其中反映了系统在遭受网络攻击时的状态变化。系统在正常运行的过程中同时监控网络流量以发现可能的网络攻击。当发生网络攻击时,系统进入防御状态,此时系统的正常服务可能会受到限制。

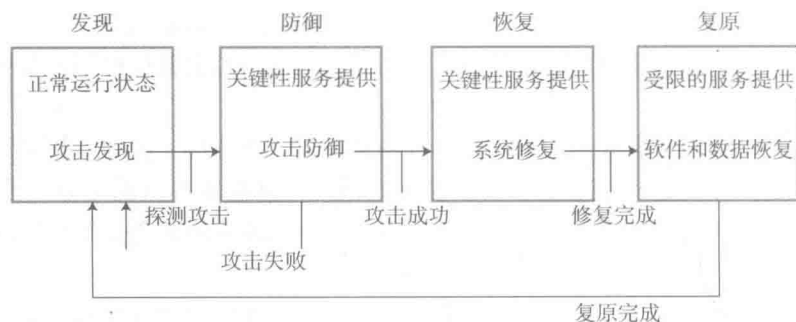


图 14-1 韧性活动

如果系统防御措施成功地击退了攻击，那么系统恢复正常服务。否则系统将进入恢复状态，此时系统将只提供关键性服务，并对网络攻击造成的损害进行修复。随后，当修复完成后系统进入复原状态，此时系统的服务逐渐恢复。最终，当所有恢复完成后，系统恢复正常服务。

正如阿波罗 13 的例子中所反映的，韧性无法预先在系统中实现。预测到所有可能出现的错误以及预测所有可能出现问题的外部环境是不可能的。因此，韧性的关键是灵活性以及适应性。正如在 14.2 节中所提到的，系统操作人员和管理人员应当有可能采取行动来保护和修复系统，即使这些行动是非常规的或者在一般情况下是不被允许的。

提高系统的韧性当然意味着较高的投资，例如，购买或修改所需要的软件，在硬件或云服务上付出额外的投资以提供在系统失效时可以使用的备份系统。这些投资的收益无法计算，因为由于系统失效或攻击导致的损失只能在发生之后才能计算出来。

因此，如果从未遭遇过严重的攻击或承受由此造成的损失的话，企业可能会不愿意在加强系统韧性上进行投资。然而，随着越来越多的破坏商业和政府系统的网络攻击事件见诸报端，对于系统韧性的需要也逐渐被广泛接受。很清楚的是，由此带来的损失可能会相当惊人，遭受攻击后业务系统有可能会无法幸免。因此，关于韧性工程的投资越来越多，以减少与系统失效有关的业务风险。

14.1 网络安全

我们整个社会面临的最严峻的问题之一就是保持我们网络化基础设施、政府、业务以及个人计算机系统的信息安全。互联网的普及以及我们对于计算机系统日益增强的依赖性为盗窃和扰乱社会秩序分子创造了新的犯罪机会。由于网络犯罪导致的损失难以度量。然而，据估计 2013 年全球经济由于网络犯罪导致的损失高达 1000 ~ 5000 亿美元（InfoSecurity 2013）。

如第 13 章所述，网络安全是一个比系统信息安全工程更加广泛的话题。软件信息安全工程主要是一个技术活动，关注保证应用系统信息安全的技术和科技。网络安全是一个社会技术关注点，涵盖了对公民、业务以及关键基础设施的保护，使其免受由于计算机和互联网的使用而导致的威胁的所有方面内容。虽然技术问题很重要，但只依赖技术很难保证信息安全。导致信息安全问题的因素包括：

- 组织策略上忽视该问题的严重性；
- 糟糕的设计以及没有严格执行信息安全规程；

- 人为疏忽；
- 对于可用性和信息安全性不恰当的权衡。

网络安全与一个组织从网络到应用系统的所有信息资产都相关。这些资产中绝大多数都是从外部采购的，企业对它们的详细运行方式并不完全理解。像 Web 浏览器这样的系统都是大型、复杂的程序，不可避免地会包含可能成为漏洞来源的软件 bug。一个组织中不同的系统以多种不同的方式相互关联。这些系统可能会在同一块磁盘上存储、共享数据、依赖于通用的操作系统构件等。组织的“系统之系统”的复杂程度令人难以置信。对于这样的复杂系统不可能保证完全没有信息安全漏洞。

因此，一般情况下都应该假设你的系统易受网络攻击，而且在某些阶段很容易发生网络攻击。一次成功的网络攻击可能会对业务造成非常严重的经济影响，因此限制攻击以及降低损失就变得尤为重要了。组织以及系统层次上有效的韧性工程可以防御攻击，并能在攻击发生后快速恢复系统正常运行，从而降低损失。

第13章中介绍了信息安全工程，其中介绍了一些对于韧性规划非常重要的思想。其中一些思想如下：

1. 资产，必须进行保护的系统和数据。一些资产比其他资产价值更高，因此需要更高等级的保护。
2. 威胁，通过破坏或窃取组织的信息技术基础设施或系统资产，从而造成损失的情形。
3. 攻击，攻击者试图破坏或窃取信息技术资产（例如网站或个人数据）的一个威胁的表现。

韧性规划过程中必须考虑的3类威胁如下。

1. 对资产机密性的威胁。在这种情况下，数据没有被破坏，但是被本来不应该具有访问权的人获得。机密性威胁的一个例子是，一家企业拥有的信用卡数据库被盗，其中的信用卡信息可能会被非法使用。
2. 对资产完整性的威胁。在这些威胁中，网络攻击通过某种方式破坏系统或数据，其中可能涉及感染软件病毒、蠕虫或者破坏组织数据库。
3. 对资产可用性的威胁。这些威胁目的是拒绝授权用户使用相关资产。最著名的例子是拒绝服务攻击，它的目的是使网站瘫痪并使外部用户无法使用。

这些威胁种类不是独立的。一个攻击者可能会通过引入恶意软件（例如僵尸网络构件）来威胁一个用户的系统完整性。该构件接下来会被远程调用，成为一个针对其他系统的分布式拒绝服务攻击的一部分。其他种类的恶意软件可以被用来获取个人信息，从而导致机密资产被非法访问。

为了对抗这些威胁，组织应该进行控制以使得攻击者很难访问或破坏资产。提升对网络安全的认识也很重要，这将使人们知道这些控制为什么重要，从而不太会向攻击者暴露信息。

可以使用的安全控制的例子如下。

1. 身份认证，一个系统的用户必须表明他们被授权访问该系统。我们所熟悉的基于登录和密码的方法是一种普遍使用的身份认证手段，但这种控制很弱。
2. 加密，数据通过算法进行混淆，从而使非授权的读取者无法访问其中的信息。许多企业现在都要求对笔记本硬盘进行加密。如果笔记本丢失或被盗，那么这一手段将降低机密信息泄漏的可能性。

3. 防火墙, 对进入系统的网络包进行检查, 根据一系列组织定义的规则接受或拒绝网络包。防火墙可以用来保证只有来自可信来源的网络流量被允许从外部互联网传入本地组织的网络。

组织中的各种控制手段构成了一个分层的保护系统。一个攻击者为了成功实施攻击必须攻破所有这些保护层。然而, 保护和效率之间需要权衡。随着保护层的增加, 系统会变得越来越慢。保护系统所耗费的存储和处理器资源越来越多, 使得用于完成有用工作的资源变少。信息越安全, 对用户越不方便, 因此他们也就更容易采取不安全的实践方式来提高系统的可用性。

与系统可依赖性的其他方面一样, 防范网络攻击的根本手段依赖于冗余和多样性。冗余意味着系统中有空闲的能力和重复的资源。多样性意味着使用不同类型的设备、软件和规程, 使得同样的失效不太会在一系列系统中同时发生。通过冗余和多样性实现网络韧性的例子如下。

1. 对于每个系统, 在多个独立的计算机系统上维护多个数据和软件的拷贝。如果可能应当尽量避免使用共享磁盘。这一点可以支持网络攻击后成功的恢复(恢复和复原)。

2. 多阶段多样化的身份认证可以防止密码攻击。除了登录/密码身份认证, 还可以包括一些附加的验证步骤, 要求用户提供一些个人信息或者由他们的移动设备生成的验证码(防御)。

3. 关键服务器可以具备冗余能力, 也就是说它们可以配备比处理所预期的负载更加强大的能力。这些空闲的能力意味着这些服务器可以在不需要降低服务器正常响应时间的情况下防御攻击。此外, 如果其他服务器被破坏, 这些空闲能力可以在其他服务器进行修复时被用于运行这些服务器上的软件(防御和恢复)。

网络安全规划必须基于资产、控制以及韧性工程的4R, 即发现(recognition)、防御(resistance)、恢复(recovery)、复原(reinstatement)。图14-2显示了一个可以遵循的规划过程, 其中的关键阶段如下。

1. 资产分类。针对组织的硬件、软件和人力资产进行检查, 并按照它们对于系统正常运行的重要程度进行分类。可以按照关键的、重要的或者有用的对它们进行分类。

2. 威胁识别。对于这些资产中的每一个(或者至少那些关键的和重要的资产), 应该识别对该资产的威胁并进行分类。某些情况下, 可以尝试对威胁发生的可能性进行评估, 但这种评估经常都不太准确, 因为缺少关于潜在攻击者的足够信息。

3. 威胁发现。对于每个威胁, 或者有时候该考虑资产/威胁对, 即考虑基于该威胁的攻击可以如何发现。你可能会需要额外购买或开发一些软件来进行威胁发现, 或者落实常规的检查规程。

4. 威胁防御。对于每个威胁或资产/威胁对, 应该考虑可能的防御策略。这些策略可能嵌入在系统中(技术策略), 也可能依赖于运行规程。你还可能要考虑威胁抵销策略, 使得威胁不再出现。

5. 资产恢复。对于每个关键性的资产或资产/威胁对, 应该考虑资产在成功的网络攻击发生后应该如何恢复。这可能会包括使用额外的硬件或者改变备份规程, 以使得访问冗余的数据拷贝更容易。

6. 资产复原。这是一种更加全面的资产恢复过程, 其中会定义恢复系统正常运行的规程。资产复原应该针对所有的资产进行考虑, 而不仅仅是那些对组织而言关键性的资产。

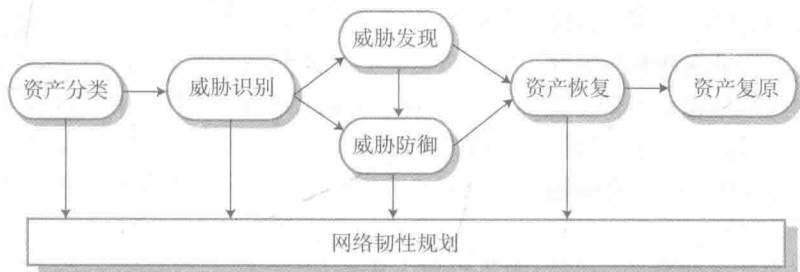


图 14-2 网络韧性规划

网络韧性规划中应该维护关于所有这些阶段的信息。该规划应该定期进行更新，并且在所有可能的地方，都应该使用针对系统的模拟攻击来对所确定的策略进行测试。

网络韧性规划另一个重要的部分是，确定如何在发生网络攻击后支持灵活的响应策略。矛盾的是，韧性和信息安全需求经常有冲突。信息安全的目标通常是尽可能限制特权以使得用户只能做那些组织的信息安全政策所允许的事情。然而，为了解决问题，用户或系统操作人员可能不得不更加主动地采取一些正常情况下应该由具有更高特权的人执行的行动。

例如，一个医疗系统的系统管理者正常情况下是不允许修改医护人员访问医疗记录的权限的。出于信息安全的考虑，访问许可必须得到正式授权，并且需要两个人一起参与权限修改。这一规定降低了系统管理人员与攻击者合谋并授予其访问机密医疗信息访问权限的机会。

现在，设想该系统管理人员注意到一个已登录的用户正在访问大量的记录，而此时并不是他的工作时间。该管理人员怀疑有账户被盗，而访问这些记录的并不是授权用户本人。为了降低破坏性，用户的访问权限应该被收回，并且对该授权用户进行检查以确定这些访问是不是非法的。然而，信息安全规程限制了系统管理人员修改用户许可的权力，使得管理人员无法采取这些措施。

韧性规划应当考虑这类情形。一种实现方式是在系统中包含一个“应急”模式，在此模式下常规检查都可以被忽略。这样，系统不再禁止这些操作，而是通过日志记录所做的事情以及为此负责的人。因此，可以使用应急处理的审计跟踪来检查系统管理人员的处理是否合理。当然，这其中也存在滥用的可能，应急模式的存在本身也是一种潜在的漏洞。因此，组织必须针对在系统中增加支持韧性的附加特性的潜在损失和收益进行权衡考虑。

14.2 社会技术韧性

从根本上说，韧性工程是一个社会技术活动而不是纯技术活动。如在第10章所解释的那样，一个社会技术系统包括硬件、软件和人，并且受到拥有和使用该系统的组织的文化、策略、规程等方面的影响。为了设计一个韧性系统，必须考虑社会技术系统设计而不是仅关注软件。韧性工程关注可能导致系统失效的不利的外部事件。在更广阔的社会技术系统范围内处理这些事件经常会更容易也更有效。

例如，Mentcare系统维护机密的病人数据，而一个可能的外部网络攻击可能会意图窃取这些数据。技术防护手段（如身份认证和加密等）可以用来保护数据，但如果攻击者能够拿到一个真实系统用户的凭据那么这些手段就无效了。你可以试图通过更复杂的身份认证规程在技术层面上解决这一问题。但是，这些规程对用户造成了不便，而且可能会导致漏洞，因为他们会用笔记下身份认证信息。一个更好的策略可能是引入组织政策和规程来强调不要

共享登录凭据的重要性，并告诉用户一些简单的方法来创建和维护更强的密码。

韧性系统具有灵活性和适应性，因此可以应对非预期的情形。开发具有适应性、能够应对非预期问题的软件是很难的。然而，正如我们从阿波罗 13 的事故中所看到的，人很善于处理这种局面。因此，为了获得韧性，应该利用人是社会技术系统的固有部分这一点。不要试图预测并处理软件中的所有问题，而应该将一些的问题留给负责操作及管理软件系统的人来解决。

为了理解为什么应该将一些问题留给他人来解决，你必须考虑社会技术系统的层次，其中包括技术系统以及软件密集型系统。图 14-3 中描述了属于一个更大范围内的社会技术系统 ST1 的技术系统 S1 和 S2。该社会技术系统包括监控 S1 和 S2 状况以及采取行动解决这些系统中的问题的操作人员。如果系统 S1 失效，那么 ST1 中的操作人员可以发现该失效并在软件失效导致更大范围内的社会技术系统失效之前采取恢复措施。操作人员还可以依照恢复和复原规程来使 S1 恢复正常运行状态。

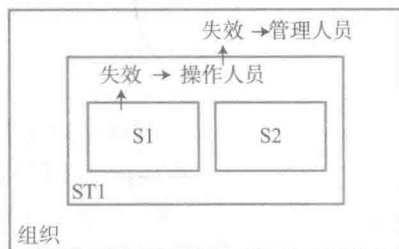


图 14-3 嵌套的技术和社会技术系统

运行和管理过程是组织和所使用的技术系统之间的接口。如果这些过程设计良好的话，那么它们能够使人们发现并处置技术系统失效，并最大限度地减少操作人员的错误。正如 14.2.2 节中所讨论的那样，采用过度自动化的严格过程并不能天然实现韧性。这种方式不允许人使用他们的技能和知识对过程进行调整和修改以应对未预计的局面和处理非预期的失效。

系统 ST1 是组织中一系列社会技术系统中的一个。如果该系统操作人员无法处置一个技术系统失效，那么这可能会导致社会技术系统 ST1 中的失效。于是，组织层的管理人员必须发现该问题并采取措施来进行恢复。因此，韧性既是一个组织特性也是一个系统特性。

Hollnagel (Hollnagel 2010) 是韧性工程的一个早期倡导者，他强调从成功以及失败中进行研究和对于组织而言十分重要。高调宣传安全信息安全失效可以引发关注，并导致实践方式和规程的改变。然而，与应对失效相比，更好的办法是通过观察人们如何应对问题并保持韧性来避免失效。这样的话，这一良好的实践可以在整个组织范围内进行传播。图 14-4 描述了 Hollnagel 所提出的反映一个组织韧性的 4 点特性。这些特性如下。

1. 应对能力。组织必须能够根据风险对自己的过程和规程进行适应性调整。这些风险可以是预测到的风险，或者是已发现的针对组织及其系统的威胁。例如，如果一个新的信息安全威胁被发现并公布，一个韧性组织能够快速做出改变以使得该威胁不会干扰组织的运行。

2. 监控能力。组织应当针对各种威胁监控自身的内部运行以及外部环境。例如，一家企业应当监控其雇员遵循信息安全政策的情况如何。如果发现潜在的不安全行为，那么企业应当做出反应，采取行动来搞清楚为什么会发生这些行为并改变雇员的行为。

3. 预见能力。一个韧性组织不能仅仅关注当前的运行，还应当对未来可能发生的影响组织运行及其韧性的事件和变化进行预见。这些事件可能包括技术创新、相关法规的变化、客户行为的变化等。例如，可穿戴技术开始变得实用，企业现在应当考虑这一技术会如何影响企业当前的信息安全政策和规程。

4. 学习能力。组织的韧性可以通过从经验中学习来提高。尤其重要的是从对不利事件的成功处置（例如对网络攻击的成功防御）中进行学习。从成功中进行学习使得好的实践可以在整个组织内得到传播。

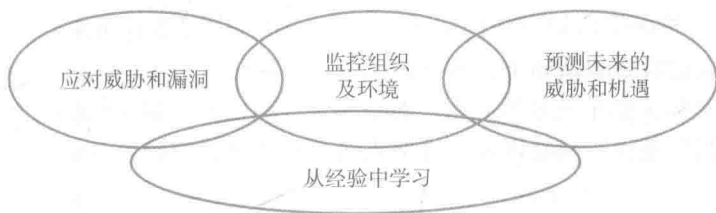


图 14-4 韧性组织的特性

如 Hollnagel 所说，想要成为韧性组织就必须在某程度上解决所有这些问题。有些组织可能会更加关注某一个质量。例如，一家运行一个大规模数据中心的企业可能最关注监控和应对能力。而一个管理长期档案信息的数字图书馆可能必须预见到未来的变化会如何影响其业务以及应对任何直接的信息安全威胁。

14.2.1 人为错误

韧性工程的早期工作关注于，安全关键系统中的事故以及操作人员的行为如何导致安全相关的系统失效。由此引发的对于系统防御的理解同样适用于需要抵挡恶意的以及偶然的人为动作的系统。

我们知道人会犯错，并且除非系统是完全自动化的，否则用户和系统操作人员有时会不可避免地做错事情。不幸的是，这些人为错误有时会导致严重的系统失效。Reason (Reason, 2000) 认为人为错误的问题可以通过以下两种角度来理解。

1. 人的角度。错误被认为是个人的责任，“不安全的动作”（例如一个操作人员未能采用一个安全性屏障）是个人的疏忽或粗心大意的行为造成的后果。持这种观点的人们认为，人为错误可以通过纪律处分的威慑力、更加严格的规程、加强训练等来减少。他们的观点是，错误是对犯错负责的个人的过失。

2. 系统的角度。基本的假设人是不可靠的并且会犯错。人们犯错是因为：他们由于高强度的工作负担而感到压力，他们未受到良好的培训，或者不合适的系统设计。好的系统应当认识到人为错误的可能性并内置相关的屏障（barrier）和保护措施以发现人为错误，并且系统可以在失效之前进行恢复。当一个失效发生后，避免其再次发生最好的办法是思考系统的防护措施如何以及为什么未能阻止错误发生。责备和惩罚触发失效的人无益于长期的系统安全。

笔者认为系统的角度是正确的观点，系统工程师应当假设系统运行过程中人为错误总是会发生。因此，为了提高一个系统的韧性，设计者必须考虑针对人为错误的防护和屏障，并将其作为系统的一部分。他们还应当考虑是否应当将这些屏障构造到系统的技术构件中。如果不可行，那么可以将这些屏障作为系统使用过程、规程以及指南的一部分。例如，要求两个操作人员来检查关键性系统的输入。

针对人为错误的屏障和防护措施可以是技术性的也可以是社会技术性的。例如，对所有的输入进行确认的代码是一种技术性的防护；针对关键性系统的更新所制定的要求两个人来确认更新的审批规程是一种社会技术性的防护。使用多样性的屏障意味着出现同样漏洞的可能性会降低，而用户错误更有可能在系统失效前得到处置。

总的来说，应该使用冗余和多样性来构建一组防护层（见图 14-5），其中每一层都使用不同的方法来防止攻击者或者处置构件失效或人为错误。深蓝色的屏障是软件检查；浅蓝色屏障是由人来执行的检查。

下面是一个这种深度防护方法的例子。作为一个空中交通管制系统中一部分的管制人员，对其人为错误的检查包括如下这些方面。

1. 将碰撞警报作为空中交通管制系统的一部分。当一个管制人员指示一架飞机改变速度或高度，系统对其轨迹进行判断以确定这架飞机与其他飞机的轨迹是否会交叉。如果是的话则要响起警报。

2. 正式的空中交通管理的记录规程。空中交通管制系统通过一个清晰定义的规程来规定如何记录发给飞机的所有控制指令。这些规程有助于管制人员来检查他们是否正确地发出了指令并且让其他人可以看到这些信息并进行检查。

3. 协作检查。空中交通管制包含一个管制人员团队，他们不断地相互检查别人的工作。当一个管制人员犯错时，其他管制人员通常会在事件发生之前发现并纠正问题。

Reason (Reason 2000) 借鉴了关于人为错误如何导致系统失效的理论中的防护层的思想。他引入了所谓的“瑞士奶酪模型”，该模型认为防护层并不是坚固的屏障而是像瑞士奶酪的切片一样。有些类型的瑞士奶酪（例如埃曼塔尔奶酪）上面有些大小不同的“孔洞”。Reason 认为漏洞或者他所称的这些层上的潜在隐患与这些“孔洞”很相似。

这些潜在隐患不是静态的——它们会随着系统的状态以及参与系统运营的人的不同而发生变化。继续我们的瑞士奶酪类比，那些“孔洞”会在系统运行期间改变大小并在防护层上发生移动。例如，如果一个系统依赖于操作人员相互检查别人的工作，那么一个可能的漏洞是他们会犯相同的错误。这在常规情况下不太会发生，因此在瑞士奶酪模型中这些“孔洞”比较小。然而，当系统的负载很高、两个操作人员工作量都很大的时候，他们犯错的可能性会增加。此时表示这些漏洞的“孔洞”会变大。

带有分层防护的系统可能会在一些可能导致破坏的外部触发事件发生时出现失效。该事件可能是人为错误 (Reason 称之为主动失效)，也可以是网络攻击。如果所有的防护屏障都失效，那么系统整体就会失效。从概念上讲，这一点对应于排列在一起的瑞士奶酪切片上的“孔洞”(如图 14-6 所示)。

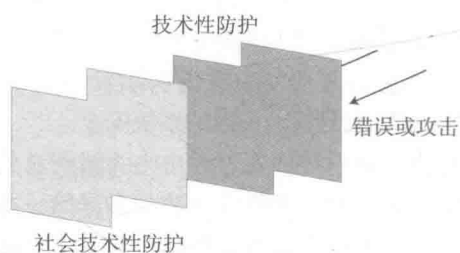


图 14-5 防护层

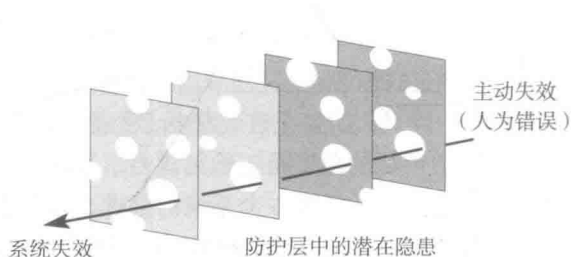


图 14-6 Reason 的系统失效瑞士奶酪模型

该模型提出可以用下面这些不同的策略来提高系统面对不利的外部事件的韧性。

1. 降低可能触发系统失效的外部事件发生的概率。为了减少人为错误，可以对操作人员进行更好的培训，或者更好地控制操作人员的工作负担以保证他们不会负担过重。为了减少网络攻击，可以减少拥有系统特权信息的人的数量，从而减少向攻击者暴露信息的机会。

2. 增加防护层的数量。一个通用法则是，在系统中的防护层越多，这些层上的“孔洞”正好对齐并导致系统失效的可能性越低。然而，如果这些防护层之间不独立，那么它们可能会都有同样的漏洞。这样的话，那些防护层就很有可能在同样的位置上有同样的“孔洞”，

因此增加新的层的好处很有限。

3. 设计系统时将各种不同类型的屏障包含其中。这意味着那些“孔洞”很大程度上会位于不同的位置，因此这些“孔洞”正好对齐并导致无法处理某个错误的可能性会降低。

4. 尽可能减少系统中潜在隐患的数量。这意味着可以有效减少系统“孔洞”的数量和大小。然而，这可能会极大地增加系统的成本。减少系统中缺陷（bug）的数量会增加测试以及验证 / 确认（V & V）成本。因此，这一选项在成本效益上可能并不好。

在设计系统时，需要考虑所有这些选项，并针对哪一个系统防护改进手段的成本效益最好做出选择。如果你在构建定制化软件，那么使用软件检查来增加防护层的数量和多样性可能是最佳选择。然而，如果你在使用成品软件，那么可能不得不考虑增加社会技术防护手段。你可以决定修改培训规程来降低发生问题的可能性，并使得问题发生时的处理更加容易。

14.2.2 运行和管理过程

所有的软件系统都有与之相关的反映设计者关于系统使用方式的假设的运行过程。有些软件系统，特别是那些控制特殊设备或者与之存在接口的软件系统，都将受过训练的操作人员作为控制系统的固有部分。在系统设计阶段要决定哪些功能应该作为技术系统的一部分，哪些功能应该属于操作人员的职责。例如，在一个医院的影像系统中，操作人员可能具有在图像处理完之后立即检查图像质量的职责。这一检查使得出现问题的时候可以重复一次成像过程。

运行过程是按照预定目的使用系统的过程。例如，空中交通管制系统的操作人员在飞机进入和离开空域、改变高度或速度、出现紧急状况等情况下都要遵循相关的过程。对于新系统而言，它们的运行过程必须在系统开发过程中进行定义和文档化。还可能要对操作人员进行培训并对其他工作过程进行调整，以使得新系统可以得到有效的使用。

然而，对于大多数软件系统而言，没有受过训练的操作人员，但是有系统用户，这些用户使用系统作为他们工作的一部分或者支持他们的个人兴趣。对于个人化系统，设计者可以描述期望的系统使用方式但却无法控制用户实际上如何使用。然而，对于企业 IT 系统而言，可以通过用户培训来教授他们如何使用系统。虽然用户行为不可控，但可以合理地期望他们一般都会遵循既定的过程。

企业 IT 系统通常还会有负责系统维护的系统管理人员。虽然他们不属于系统所支持的业务过程的一部分，但是他们的工作是监控软件系统的错误和问题。如果发生问题，系统管理人员采取行动来解决问题并将系统恢复为正常运行状态。

在此之前，讨论了深度防护以及使用多样化的机制来检查可能导致系统失效的不利事件的重要性。运行和管理过程是一种重要的防护机制，在设计过程时需要在高效的运行和问题管理之间寻求一种平衡。如图 14-7 所示，它们经常有冲突，因为提高效率需要降低系统的冗余度和多样性。

高效的过程运行	问题管理
过程优化和控制	过程灵活性和适应性
信息隐藏和信息安全	信息共享和可见性
通过自动化来降低操作人员工作负担，并减少操作人员和管理人员数量	通过人工过程以及备用的操作人员 / 管理人员能力来应对问题
角色专业化	角色共享

图 14-7 效率和韧性

在过去的 25 年中, 很多人关注所谓的过程改进。为了提升运行和管理过程的效率, 企业研究他们的过程执行情况并寻找特别高效和低效的实践。对高效的实践进行编纂和文档化, 还可能会开发软件来支持这一“最优的”过程。低效的实践被更加高效的做事情的方式所取代。有时候还会引入过程控制机制来保证系统操作人员和管理人员遵循这一“最佳实践”。

过程改进的问题是它经常使人更难处理问题。有些看起来“低效”的实践经常是因为人们想要保持冗余的信息或共享信息, 因为他们知道这会使出现问题时应对起来更容易。例如, 空中交通管制人员也可以依赖航班数据库打印航班详细信息, 因为这样的话, 即使系统数据库不可用他们也有航班的信息。

人具有有效应对非预期情形的独特能力, 即使他们从来都没有应对这类情形的直接经验。因此, 当出问题时, 操作人员和系统管理人员经常可以恢复正常运行, 虽然他们有时可能会打破常规并使用一些“变通”的过程。因此, 应该将运行过程设计得具有灵活性和适应性。运行过程不应该约束太多, 不应该要求按照某种顺序执行操作, 而系统软件不应该依赖于对特定过程的遵循。

例如, 一个应急服务控制室系统被用于管理应急呼叫以及对这些呼叫的应答。处理一个呼叫的“常规”过程是记录呼叫者的详细信息, 然后向相应的应急服务发送一条消息并提供事件的详细信息和地址。这一规程提供了所采取的行动的审计轨迹。随后的调查可以检查应急呼叫是否得到了适当的处理。

现在设想该系统受到了拒绝服务攻击, 这使得消息系统不可用。操作人员并没有简单地忽略这些呼叫, 而是使用他们自己的个人移动电话以及他们对于应急服务的知识来直接呼叫相应的应急服务单位, 使得他们可以及时响应严重事故。

信息的管理和提供对于韧性运行也是很重要的。为了使一个过程更加高效, 有必要在操作人员需要的时候提供他们所需要的信息。从信息安全的角度看, 除非操作人员或管理人员需要相关信息, 否则不应该允许访问这些信息。然而, 更加自由的信息访问方法可以改进系统的韧性。

如果操作人员只能得到过程设计者认为他们“需要知道”的信息, 那么他们可能无法发现那些不直接影响他们手头任务的问题。当问题出现时, 系统操作人员无法获得对于系统中所发生的事情的全面了解, 因此他们更加难以制定问题处理的策略。如果他们由于信息安全的原因无法访问系统中的一些信息, 那么他们可能无法阻止攻击并修复已导致的破坏。

自动化系统管理过程意味着一个管理人员可能就能管理大量的系统。自动化系统可以发现通用的问题并采取行动以使系统从问题中恢复。系统运行和管理所需的人更少, 因此成本可以降低。然而, 过程自动化有如下两个不利之处。

1. 自动化的管理系统可能会出错并采取不正确的动作。随着问题的发展, 系统可能会采取意想不到的动作, 系统管理人员无法理解这些动作, 而且会使得情况变得更糟糕。

2. 协作过程中的问题解决。如果管理人员很少, 那么制定问题或网络攻击恢复策略的时间很可能会更长。

因此, 过程自动化对于系统韧性可能既有正面的影响又有负面的影响。如果自动化系统正常工作, 那么它可以发现问题, 在需要的时候调用网络攻击防御手段, 并启动自动的恢复过程。然而, 如果自动化系统无法处理问题, 应对问题的人会不够, 而系统可能会被做出错误动作的过程自动化所破坏。

在一个包含不同种类的系统和设备的环境中,期望所有操作人员和管理人员都能够应对所有这些不同的系统可能并不现实。因此,个人专业化可以使得他们成为一小部分系统的专家。这导致了更高效的系统运行,但是会对系统的韧性有所影响。

角色专业化的问题是,在某个特定时间点可能没人理解系统之间的交互。其结果是,如果专家不在那么就很难应对问题。如果人们同时在多个系统上工作,那么他们可以理解系统之间的依赖和关系,因此也就可以处理影响多个系统的问题。如果没有专家,处理问题以及修复问题所导致的破坏会变得困难得多。

如第13章中所提到的,可以使用风险评估来帮助你做出过程效率和韧性之间的平衡决策。考虑所有可能需要操作人员或管理人员进行干预的地方所蕴含的风险,评估这些风险的可能性以及可能导致的损失的程度。对于可能导致严重破坏以及大量损失的风险以及很可能发生的风险,应该更加强调韧性而不是过程效率。

14.3 韧性系统设计

韧性系统可以防御不利的事件(例如,软件失效和网络攻击)并从中恢复。它们以最小的干扰提供关键性服务,并在问题发生后快速恢复正常运行状态。在设计一个韧性系统时,必须假设系统失效或者攻击者入侵会发生,必须考虑通过冗余以及多样性的特征来应对这些不利事件。

韧性系统设计包括两个密切相关的工作流。

1. 识别关键性服务和资产。关键性服务和资产是那些使系统可以满足其主要目的的系统元素。例如,一个系统的主要目的是处理响应紧急呼叫的救护车派遣以使得需要帮助的人尽快得到帮助。这里的关键性服务包括接听呼叫并派遣救护车进行医疗急救。其他服务,例如呼叫日志记录以及救护车追踪则没那么重要。

2. 设计支持问题发现、防御、恢复和复原的系统构件。例如,在救护车派遣系统中,可能要包含一个监控者定时器(见第12章)来侦测系统是否对事件没有响应。操作人员可能要通过硬件令牌进行身份认证以防止非授权的访问。如果系统失效,那么呼叫可以被转移到另一个中心以保持基本服务的正常运行。可能要在不同硬件上部署系统数据库和软件的拷贝以使得系统可以在中断运行后复原。

基本的发现、防御、恢复的思想是 Ellison 等人 (Ellison et al. 1999, 2002) 在韧性工程中的早期工作的基础。他们设计了一种分析方法,叫作可生存性系统分析 (survivable system analysis)。这种方法被用于评估系统中的漏洞、支持系统体系结构以及提升系统可生存性的特性的设计。

可生存性系统分析是一个四阶段过程(图14-8),该过程分析当前或者所提出的系统需求和体系结构,识别关键性服务、攻击场景以及系统“脆弱点”,并提出改进系统可生存性的手段。其中每个阶段中的关键活动如下:

1. 系统理解。对于一个已有的或所提出的系统,评审系统目标(有时称为使命

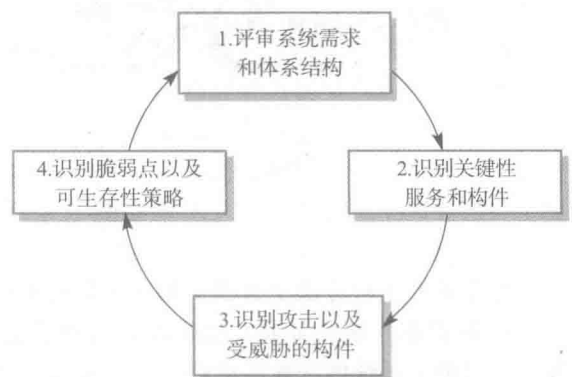


图 14-8 可生存性分析中的阶段

目标)、系统需求以及系统体系结构。

2. 关键性服务识别。识别必须永远保持的服务, 以及维持这些服务所需要的构件。

3. 攻击模拟。识别可能的攻击场景或用况, 以及可能受到这些攻击影响的系统构件。

4. 可生存性分析。识别重要的并且受到威胁的构件, 基于问题防御、发现和恢复来识别可生存性策略。

这种可生存性分析方法的根本问题是, 它的起点是系统的需求和体系结构文档。这一点对于国防系统(美国国防部资助的项目)可能是合理的假设, 但对于业务系统而言存在以下两个问题。

1. 它与针对韧性的业务需求并不是明确相关的。笔者认为这些需求比技术系统需求作为起点更合适。

2. 它假设存在详细的系统需求陈述。而实际上, 必须实现韧性的系统可能没有完整的或最新的需求文档。对于新系统而言, 韧性自身可能就是一个需求, 或者系统可能要用敏捷方法来进行开发。系统体系结构设计可以考虑韧性。

一个更加通用的韧性工程方法(如图 14-9 所示)考虑了缺少详细需求, 以及在系统中明确地设计恢复和复原机制的情况。对于一个系统中的大多数构件, 你都没有它们的源代码也无法对它们进行修改。你的韧性策略必须在考虑这些限制条件的情况下进行设计。

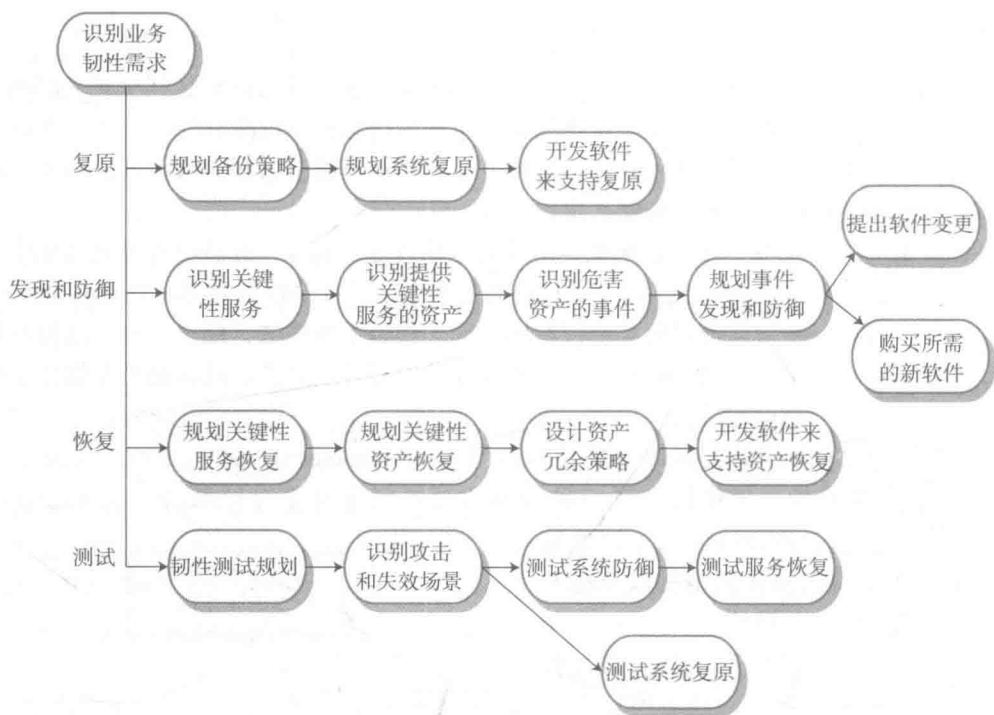


图 14-9 韧性工程

这个韧性工程方法中有 5 个相互关联的工作流。

1. 识别业务韧性需求。这些需求明确了业务作为一个整体如何保持它提供给客户的服务, 针对单个系统的韧性需求将会在此基础上开发得到。实现韧性是很昂贵的, 没有必要对系统韧性进行过度设计以提供不必要的韧性支持。

2. 规划如何在不利的事件发生后将一个或一组系统复原为它们的正常运行状态。这一规划必须与业务的常规备份和归档策略（它们可以支持技术性错误或人为错误后的信息恢复）相集成。该规划还应该成为更大范围内的灾难恢复策略的一部分。必须考虑物理事件（例如火灾和水灾）的可能性，并考虑如何在不同的地点保存关键性信息。可以决定针对该规划使用云备份。

3. 识别可能危害系统的失效以及网络攻击，并且设计相应的发现和韧性策略来应对这些不利事件。

4. 规划如何在关键性服务由于失效或网络攻击而导致破坏或下线后对其迅速恢复。这一步骤通常包括准备这些服务的关键性资产的冗余拷贝，并在需要时切换到这些拷贝上。

5. 重要的是，应当测试韧性规划的所有方面。这一测试包括识别失效和攻击场景，并针对系统测试这些场景。

保持关键性服务的可用性是韧性的本质。相应地，你必须了解：

- 对于业务最关键的系统服务；
- 必须保持的最低的服务质量；
- 这些服务将如何受到危害；
- 这些服务可以如何进行保护；
- 如果服务不可用，那么该如何快速恢复它们。

作为关键性服务分析的一部分，必须识别那些对于提供这些服务起着关键作用的系统资产。这些资产可以是硬件（服务器、网络等）、软件、数据和人。为了构建一个韧性系统，必须考虑如何使用冗余和多样性来保证这些资产在系统失效时也能保持可用性。

对于所有这些活动，提供不利事件发生后的快速响应和恢复计划的关键是，有额外的软件来支持防御、恢复和复原。这些软件可以是商业的信息安全软件，或者是在应用软件中编程实现的韧性支持，其中还可能包括用于恢复和复原的脚本以及特别编写的程序。如果你有正确的支持软件，恢复和复原过程可以部分自动化并在系统失效后被快速调用和执行。

韧性测试包括模拟可能的系统失效和网络攻击来测试所制订的韧性计划是否按照预期工作。测试很重要，因为我们从经验中知道韧性规划过程中所做的假设经常是不正确的，而且所规划的行动并不总能起到作用。针对韧性的测试可以揭示这些问题，从而使得韧性计划可以得到细化。

测试可能非常困难和昂贵，因为很显然测试不能在一个运行系统上进行。系统及其环境可能要被复制用于测试，而工作人员可能不得不从他们的常规职责中脱身出来，以便在测试系统上工作。为了降低成本，可以使用“桌面测试”。测试团队假设问题已经发生，测试他们对于问题的反应；他们并不会在真实系统上模拟问题。虽然这种方法可以提供关于系统韧性的有用信息，但它在发现韧性规划中的不足方面不如测试那么有效。

作为该方法的一个例子，让我们看一下针对 Mentcare 系统的韧性工程。回顾一下，这个系统支持临床医生在不同的地点处置有心理健康问题的病人。该系统为医生和专科护士提供病人信息以及诊疗记录。该系统包括一些检查，可以标记出一些可能具有潜在危险或自杀倾向的病人。图 14-10 描述了该系统的体系结构。

医生和护士在诊疗之前以及过程中使用该系统，诊疗结束后病人信息会进行更新。为了确保临床有效性，其中的业务韧性需求是关键性的系统服务要在常规工作时间保持可用，系统失效或网络攻击不会导致病人数据永久损坏或丢失，病人信息不应该开放给未授权的人。

该系统中有两个关键性服务必须保持。

1. 一个信息服务, 提供关于一个病人当前诊断和治疗计划的信息。

2. 一个警告服务, 特别指出可能对他人或患者自身造成危险的病人。

注意关键性服务并不是完整的病人记录的可用性。医生和护士只需要偶尔看一下过去的治疗记录, 因此如果完整记录不可用, 那么临床护理并不会受到严重影响。因此, 使用一个仅包含病人以及近期治疗信息的概述记录可能就能提供有效的护理。

在常规系统运行中提供这些服务所需的资产包括:

1. 保存所有病人信息的病人记录数据库;
2. 为本地客户端计算机提供数据库访问的数据库服务器;
3. 客户端/服务器通信所用的网络;
4. 临床医生用于访问病人信息的本地笔记本或桌面计算机;
5. 识别具有潜在危险的病人并标记病人记录的一组规则, 客户端软件向系统用户重点提示危险的病人。

为了规划发现、防御和恢复策略, 需要开发一组预见到可能危害系统所提供的关键性服务的不利事件的场景。这些不利事件的例子包括:

1. 由于系统失效、网络失效或拒绝服务网络攻击造成的数据库服务器不可用;
2. 病人记录数据库或定义“危险病人”含义的规则故意或意外损坏;
3. 客户端计算机感染恶意软件;
4. 未授权的人访问客户端计算机并获得了病人记录访问权限。

图 14-11 描述了这些不利事件可能的发现和防御策略。注意这些不仅仅是技术方法, 还包括面向系统用户介绍信息安全问题的培训班。我们知道许多违反信息安全的行为之所以出现, 是因为用户无意中向一个攻击者泄漏了特权信息, 这些培训减少了这种情形发生的机会。这里不用更多的篇幅来讨论图 14-11 中列出的所有选项, 这里关注如何通过修改系统体系结构使系统具有更好的韧性。

图 14-11 建议, 在客户端计算机上保存病人信息是一种可以帮助维持关键性服务的冗余策略。这导致了图 14-12 中所示的修改后的软件体系结构。该体系结构的关键特性包括:

1. 在本地客户端计算机上保存概要病人记录。本地计算机相互之间可以使用系统网络或者必要时使用移动电话创建的自组织网络来直接通信并且交换信息。因此, 如果数据库不可用, 医生和护士仍然可以访问重要的病人信息 (防御和恢复)。

2. 主服务器失效后的备份服务器。该服务器负责定期读取数据库的快照作为备份。当主服务器失效时, 该服务器也可以面向整个系统扮演主服务器。这一点保证了服务的持续性以及服务器失效后的恢复 (防御和恢复)。

3. 数据库完整性检查和恢复软件。完整性检查作为一个后台任务运行, 检查数据库损坏的迹象。如果发现损坏, 它可以自动启动恢复过程, 从备份那里恢复一些或全部数据。事务

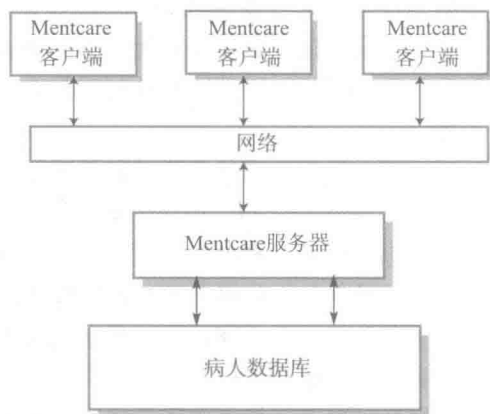


图 14-10 Mentcare 系统的客户-服务器体系结构

日志使得这些备份可以根据最近的变化进行更新（发现和恢复）。

事 件	发 现	防 御
服务器不可用	1. 使用客户端上的监控者定时器，如果对于客户端访问没有响应就产生超时 2. 系统管理者向临床用户发送文本消息	1. 设计系统体系结构来保存关键性信息的本地拷贝 2. 提供客户端之间病人数据的对等（P2P）搜索 3. 为工作人员提供智能手机，使得他们可以在服务器失效时使用手机访问网络 4. 提供备份服务器
病人数据库损坏	1. 记录级别的密码校验 2. 常规的数据库完整性自动检查 3. 向系统报告不正确的信息	1. 使用可重做的事务日志来更新数据库备份，重做最近的事务 2. 保存病人信息和软件的本地拷贝，以便从本地拷贝和备份中恢复数据库
客户端计算机感染恶意软件	1. 向系统报告使得计算机用户能够报告异常的行为 2. 在启动时进行自动的恶意软件检查	1. 面向所有系统用户进行信息安全意识培训 2. 禁用客户端计算机上的 USB 端口 3. 针对新客户端进行自动的系统启动 4. 支持使用移动设备访问系统 5. 安装信息安全软件
对病人信息的未授权访问	1. 来自用户的人侵警告文本消息 2. 针对异常活动的日志分析	1. 多级的系统身份认证过程 2. 禁用客户端计算机上的 USB 端口 访问日志以及实时的日志分析 4. 面向所有系统用户进行信息安全意识培训

图 14-11 不利事件的发现和防御策略

为了保持病人信息访问以及工作人员警告这两个关键性服务，我们可以利用客户 / 服务器系统中固有的冗余性。通过在一次临床治疗过程开始时将信息下载到客户端，诊断可以在不访问服务器的情况下继续进行。只有那些安排好当天进行诊疗的病病人的信息需要下载。如果需要访问其他病人的信息并且服务器不可用，那么可以通过对等通信联系其他客户端计算机来看是否有所需要的信息。

向工作人员提供危险病人警告的服务可以很容易地用这种方法来实现。可能危害自己或他人的病人的记录在下载过程之前识别出来。当医护人员访问这些记录时，软件可以突出显示这些记录以提醒这些病人需要特殊照顾。

该体系结构中支持不利事件防御的特性也同样支持从不利事件中的恢复。通过保存多个信息副本并且使用备份硬件，关

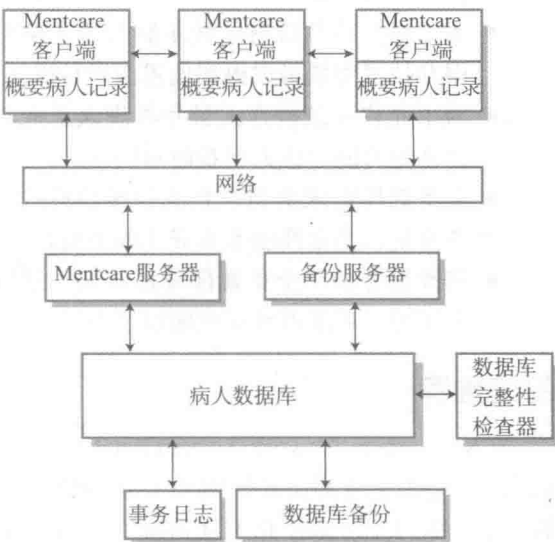


图 14-12 Mentcare 系统韧性体系结构

键性系统服务可以快速恢复正常运行。因为系统只需要在常规工作时间（即早上 8 时到晚上 6 时），该系统可以在夜里进行复原以使系统在失效后第二天可以正常使用。

除了保持关键性服务，保持病人数据的机密性和完整性的业务需求也必须支持。图 14-12 中所显示的体系结构包括一个备份系统和专门的数据库完整性检查，以便减少病人信息意外地或在恶意攻击中被破坏的机会。客户端计算机上的信息也是可用的，可以支持从数据损坏或破坏中恢复。

尽管保存多个数据拷贝是一种针对数据损坏的防护手段，它对于机密性而言是一种风险，因为所有这些拷贝都必须保证信息安全。这种情况下，该风险可以通过以下手段进行控制。

1. 仅下载那些将要进行诊疗的病人的概要记录。这限制了可能受到危害的记录数量。
2. 加密本地客户端计算机上的磁盘。没有密钥的攻击者即使获得了客户端计算机的访问权限也无法读取磁盘。
3. 在一次诊疗过程结束后安全地删除所下载的信息。这进一步降低了攻击者获取机密信息的可能性。
4. 确保所有的网络事务都是加密的。如果攻击者拦截了这些事务，他们也无法访问这些信息。

因为对性能有影响，对整个服务器上的病人数据库加密很有可能是不现实的。因此应该使用很强的验证手段来保护这些信息。

要点

- 一个系统的韧性是对该系统在干扰性事件（例如，设备失效和网络攻击）发生时能够保持系统关键性服务持续提供的能力的一种判断。
- 韧性应当基于 4R 模型，即发现（recognition）、防御（resistance）、恢复（recovery）、复原（reinstatement）。
- 韧性规划应当基于网络化系统会遭到恶意的内部和外部网络攻击并且其中一些攻击会成功的假设。
- 系统设计应当带有一些不同类型的防护层。如果这些防护层是有效的，那么人为错误和技术失效可以得到处置而且可以防御网络攻击。
- 为了允许系统操作人员和管理人员应对问题，过程应当具有灵活性和适应性。过程自动化可能会让人更难应对问题。
- 业务韧性需求应当是系统韧性设计的起点。为了实现系统韧性，必须关注问题发现和恢复、关键性服务和资产的恢复，以及系统的复原。
- 韧性设计的一个重要部分是识别关键性服务，这些服务对于系统实现其主要目的十分重要。系统设计应该使得这些服务得到保护并且在发生失效时尽快恢复。

阅读推荐

《Survivable Network System Analysis: A Case Study》是一篇很棒的论文，其中介绍了系统可生存性的思想，并使用了一个心理健康记录处理系统的案例研究来说明可生存性方法的应用。（R. J. Ellison, R. C. Linger, T. Longstaff, and N. R. Mead, IEEE Software, 16（4）, July/August 1999）<http://dx.doi.org/10.1109/52.776952>

《Resilience Engineering in Practice: A Guidebook》是一些关于采取更广泛的社会技术系统观点的韧性工程的文章和案例研究。(E. Hollnagel, J. Paries, D. W. Woods, and J. Wreathall, Ashgate Publishing Co., 2011)

《Cyber Risk and Resilience Management》是一个有大量关于网络安全和韧性的资源的网站, 包括一个韧性管理模型。(Software Engineering Institute, 2013) <https://www.cert.org/resilience/>

网站

本章的PPT: <http://software-engineering-book.com/slides/chap14/>

支持视频的链接: <http://software-engineering-book.com/videos/security-and-resilience/>

练习

- 14.1 如何使用互补的发现、防御、恢复、复原策略来实现系统的韧性?
- 14.2 如何使用冗余和多样性来提高系统防御可能损坏或破坏系统资产的网络攻击的能力?
- 14.3 韧性组织是什么意思? 解释图 14-4 中所示的 Hollnagel 模型如何成为改进组织韧性的有效基础。
- 14.4 一个医院提出引入一个新政策, 规定任何医护人员(医生或护士)如果采取某种行为或授权某种行为导致病人受伤那么将会被刑事起诉[©]。解释为什么这个想法很糟糕, 很有可能无法提高病人安全性。为什么还很有可能对组织的韧性造成负面影响?
- 14.5 针对一个信息系统, 提出可以包含在其中的 3 个防护层, 以保护数据项不会被未经授权的人所修改。
- 14.6 为什么过程不灵活会限制一个社会技术系统防御不利事件(例如, 网络攻击和软件失效)以及从中恢复的能力? 如果你有过程不灵活导致问题的经验, 利用你的经验举出一些例子来说明你的答案。
- 14.7 说一说图 14-9 中所提出的韧性工程方法如何与系统中软件的敏捷开发过程相结合一起使用。对于韧性很重要的系统, 使用敏捷开发会出现什么样的问题?
- 14.8 在 13.4.2 节中, 已经发生了: (1) 一个未授权用户下恶意订单以影响价格; (2) 一次损坏了交易数据库的入侵。对于其中每一个网络攻击, 说一说可以使用的防御、发现和恢复策略。
- 14.9 图 14-11 提出了一些可能影响 Mentcare 系统的负面事件。考虑一个针对该系统的测试计划, 确定如何测试 Mentcare 系统发现、防御这些负面事件以及从中恢复的能力。
- 14.10 一家公司的高级管理人员担心对公司不满的员工会针对公司的信息技术资产发起内部攻击。作为一个韧性改进计划的一部分, 她提出要引入一个日志系统和数据分析软件来捕捉并分析所有的雇员行为, 但雇员不应该得知这一系统的存在。讨论引入这样一个日志系统以及在不告诉系统用户的情况下这么做所引发的道德问题。

参考文献

Ellison, R. J., R. C. Linger, T. Longstaff, and N. R. Mead. 1999. "Survivable Network System Analysis: A Case Study." *IEEE Software* 16 (4): 70-77. doi:10.1109/52.776952.

Ellison, R. J., R. C. Linger, H. Lipson, N. R. Mead, and A. Moore. 2002. "Foundations of Survivable

Systems Engineering.” *Crosstalk: The Journal of Defense Software Engineering* 12: 10–15.
http://resources.sei.cmu.edu/asset_files/WhitePaper/2002_019_001_77700.pdf

Hollnagel, E. 2006. “Resilience—the Challenge of the Unstable.” In *Resilience Engineering: Concepts and Precepts*, edited by E. Hollnagel, D. D. Woods, and N.G. Leveson, 9–18.

———. 2010. “RAG—The Resilience Analysis Grid.” In *Resilience Engineering in Practice*, edited by E. Hollnagel, J. Paries, D. Woods, and J. Wreathall, 275–295. Farnham, UK: Ashgate Publishing Group.

InfoSecurity. 2013. “Global Cybercrime, Espionage Costs \$100–\$500 Billion Per Year.” <http://www.infosecurity-magazine.com/view/33569/global-cybercrime-espionage-costs-100500-billion-per-year>

Laprie, J-C. 2008. “From Dependability to Resilience.” In *38th Int. Conf. on Dependable Systems and Networks*. Anchorage, Alaska. http://2008.dsn.org/fastabs/dsno8fastabs_laprie.pdf

Reason, J. 2000. “Human Error: Models and Management.” *British Medical J.* 320: 768–770.
doi:10.1136/bmj.320.7237.768.

高级软件工程

本部分介绍一些更高级的软件工程话题。在阅读这些章节之前，需要读者已经理解了第1～9章中所介绍的软件工程基础知识。

第15～18章关注基于Web的信息系统和企业系统的开发中处于主导地位的开发范型——软件复用。第15章介绍了这个话题，并解释了可能的不同类型的复用；介绍了最常用的复用方法，即应用系统的复用，这些应用系统按照每种业务的特定要求进行配置和适应性修改。

第16章关注软件构件而不是整个软件系统的复用。这一章解释了构件是什么以及为什么有效的构件复用需要标准的构件模型；还讨论了基于构件的软件工程的一般过程以及构件组装的问题。

大部分大型系统现在都是分布式系统，第17章介绍了构造分布式系统的问题。本书将客户-服务器方法作为一种基本的分布式系统工程范型进行了介绍，解释了实现这种体系结构风格的各种方式。最后一节介绍了软件即服务（software as a service）——在互联网上交付软件功能，它已经改变了软件产品市场。

第18章介绍了与面向服务的体系结构相关的话题，它将分布式和复用的思想联系在一起。服务是可复用的软件构件，其功能可以通过互联网访问。讨论了两种广泛使用的服务开发方法，即基于SOAP的服务以及RESTful服务，解释了创建服务（服务工程）和组装服务以创建新的软件系统的过程中所包含的内容。

第19～21章的关注点是系统工程。在第19章中，介绍了该话题并解释了为什么软件工程师理解系统工程是很重要的；讨论了系统工程生命周期以及该生命周期中采购的重要性。

第20章介绍了系统之系统（systems of systems, SOS）。21世纪构造的大型系统不会从头进行开发，而是通过集成现有的复杂系统来创建。解释了为什么在开发系统之系统中理解复杂性很重要，并且讨论了复杂系统之系统的体系结构模式。

大多数软件系统并不是应用程序或业务系统，而是嵌入式实时系统，第21章中介绍了这一重要的话题。介绍了实时嵌入式系统的思想，描述了嵌入式系统设计中所使用的体系结构模式；解释了时间性分析的过程，并且以一个关于实时操作系统的讨论结束了这一章。

软件复用

目标

本章的目标是介绍软件复用并描述基于大规模软件复用的系统开发方法。阅读完本章后，你将：

- 理解开发新系统时复用软件的收益和问题；
- 理解作为一组可复用对象的应用框架的概念，以及框架可以如何在应用开发中使用；
- 了解软件产品线，它包括一组通用的核心体系结构和可复用构件，这些构件可以面向每一个产品版本进行配置；
- 学习如何通过配置和组装成品应用软件系统来开发系统。

基于复用的软件工程是一种软件工程策略，其中开发过程会复用现有的软件。直到 2000 年左右，系统化的软件复用仍然不那么普遍，但是复用现在已经在新的业务系统的开发中得到了广泛使用。更低的软件开发和维护成本、更快的系统交付、更高的软件质量驱动了向基于复用的开发的转换。企业将他们的软件视为一种有价值的资产。他们提升对现有系统的复用来提高自己在软件投资上的回报。

不同类型的可复用软件现在都广泛地存在着。开源运动使得我们有了巨大的可以复用的代码库。这些可复用代码的形式可以是程序库或者整个应用。现在许多特定领域的应用系统（例如 ERP 系统）都可以根据客户需求进行裁剪和适应性修改。一些大的软件公司为客户提供了系列可复用构件。相关标准（例如 Web 服务标准）使得开发软件服务并在一系列应用中进行复用变得更容易。

基于复用的软件工程是一种试图最大化复用现有软件的开发方法。可以复用的软件单元的大小差异很大。例如下面这些：

1. 系统复用。可能由一些应用程序组成的完整的系统，可以作为系统之系统（见第 20 章）的一部分进行复用。
2. 应用复用。一个应用可以不经修改直接纳入到别的系统中而进行复用，或者通过面向不同的客户进行配置来进行复用。此外，也可以使用应用族或者软件产品线来开发新系统，它们具有通用的体系结构，但是可以根据不同的客户需求进行适应性修改。
3. 构件复用。一个应用的构件，从规模很大的子系统到规模很小的单个对象，也可以复用。例如，作为一个文本处理系统的一部分开发的一个模式匹配系统，可以在一个数据库管理系统中进行复用。构件可以部署在云上或者私有服务器上，可以作为服务通过应用编程接口（application programming interface, API）进行访问。
4. 对象和函数复用。实现单个功能（例如数学函数）或者一个对象的软件构件也可以复用。这种形式的复用是围绕标准库设计的，在过去 40 年中一直很常见。许多可用的函数库和类库都是免费的。通过将这些库中的类和函数与新开发的应用代码链接到一起对它们进行复用。数学算法和图形这样的领域需要专业、昂贵的专家知识来开发高效的对象和函数，

此时复用尤其具有成本效益优势。

所有包含通用功能的软件系统和构件都具有潜在的可复用性。然而，这些系统或构件有时是具有特定性的，以至于修改它们以适应一个新的情形非常昂贵。除了复用代码，还可以复用作为软件基础的思想。这被称为概念复用。

在概念复用中，不是复用软件构件，而是复用一个想法、一种工作方式，或者一个算法。所复用的思想以一种抽象的表示法（例如一个系统模型）进行表示，其中不包含实现细节。因此，它可以面向一系列不同的情形进行配置和适应性修改。概念复用包含在一些方法之中，例如设计模式（见第 7 章）、可配置的系统产品、程序生成器。当概念被复用时，复用过程必须包含一个活动来对抽象概念进行实例化以创建可执行的构件。

软件复用的一个明显的优势是，总体的开发成本会更低。需要进行规格说明、设计、实现和确认的软件构件会更少。然而，降低成本只是软件复用的一个优势。图 15-1 中列出了软件复用的其他好处。

收 益	解 释
开发加速	将一个系统尽早推向市场经常比总体开发成本更重要。复用软件可以加速系统生产，因为开发和确认所需的时间都可以减少
有效的专家利用	应用专家不再需要一次又一次做同样的工作，而是开发封装了他们的知识的可复用软件
提高可依赖性	被复用的软件在工作的系统中进行了尝试和测试，因此应该比新软件具有更好的可依赖性。它的设计和实现故障应该已经被发现和修复了
降低开发成本	开发成本与所开发的软件的规模成正比。复用软件意味着需要编写的代码行更少
降低过程风险	已有软件的成本是已知的，而开发成本总是需要判断，这是一个很重要的项目管理因素，因为复用软件降低了项目成本估算中的边际误差，特别是当复用大规模软件构件（例如子系统）时尤其如此
符合标准	有些标准，例如用户界面标准，可以被实现为一组可复用构件。例如，如果一个用户界面中的菜单使用可复用构件实现，那么所有应用都会向用户呈现出同样的菜单格式。使用标准的用户界面可以提高可依赖性，因为用户在面对一个熟悉的界面时会更少犯错误

图 15-1 软件复用的收益

然而，复用也有一些相关的成本和困难（见图 15-2）。理解一个构件在一个特定情形下

问 题	解 释
创建、维护以及使用一个构件库	创建一个可复用构件库并确保软件开发者可以使用这个构件库可能会很昂贵。必须对开发过程进行调整以确保使用构件库
查找、理解以及适配可复用构件	必须能从一个库中找到软件构件、理解它，有时候还需要对其进行适配以用于一个新的环境。工程师在将构件查找作为他们常规开发过程的一部分之前，必须对于在构件库中找到想要的构件相当有信心
维护成本增加	如果没有提供一个被复用的软件系统或构件的源代码，那么维护成本可能会更高，因为被复用的系统元素可能会随着对系统的修改而变得不兼容
缺少工具支持	一些软件工具不支持基于复用的开发。将这些工具与构件库系统集成可能很难甚至不可能。这些工具所假设的软件过程可能并没有考虑复用。与面向对象开发工具相比，支持嵌入式系统工程的工具更有可能是这种情况
“不是在这里发明的”综合症	一些软件工程师喜欢对构件进行修改，因为他们认为他们可以改进这些构件。这样做部分与信任有关，同时也部分因为与复用其他人的软件相比编写原创的软件被视为更具挑战性

图 15-2 软件复用的问题

是否适合复用以及测试该构件以保证其可依赖性都有着显著的成本。这些附加的成本意味着在开发成本方面的节省可能会低于预期。然而，复用的其他收益仍然成立。

如第2章所述，软件开发过程必须进行调整以充分考虑复用。特别是，要有一个需求精化阶段来对系统需求进行修改以反映可用的可复用软件。系统的设计和实现阶段也可能要包括一些显式的活动，以便针对复用来寻找和评估候选构件。

15.1 复用概览

在过去20年中，人们开发了许多技术来支持软件复用。这些技术利用了以下几个方面的事实：同属一个应用领域的系统相似且具有复用潜力；复用可以在多个不同层次上进行，从简单的函数到完整的应用；可复用构件标准有利于复用。图15-3显示了一个“复用概览”，其中包括实现软件复用的不同方式。图15-4对这些复用方法一一进行了介绍。



图 15-3 复用概览

有了这一系列复用技术后，关键问题是“在某种特定情形下哪种技术是最适合使用的”？显然，这个问题的答案取决于所开发的系统的需求、可用的技术和可复用资产、开发团队的技术水平等。在规划复用时应当考虑以下这些关键因素。

1. 软件的开发进度。如果软件必须要快速开发，应当尽量复用完整的系统而不是单个构件。虽然可复用的完整系统对于需求的符合度可能并不完美，但是这种方法可以让所需要的开发量最小化。

2. 所期望的软件生命周期。如果开发一个长生命周期的系统，应当关注系统的可维护性。不应当只考虑复用的短期收益，而是要考虑长期的影响。在整个生命周期中，你将不得不为了实现新需求而对系统进行适应性调整，这意味着要对系统各个部分进行修改。如果你无法访问可复用构件的源代码，你可能会倾向于避免使用来自外部供应商的成品构件以及系统，因为这些供应商可能无法为所复用的软件提供持续的支持。你可能会决定复用开源系统和构件更安全一些（见第7章），因为这意味着你可以访问源代码并持有代码拷贝。

3. 开发团队的背景、技能和经验。所有的复用技术都相当复杂，需要相当多的时间来有效地理解和使用它们。因此，应当将复用的注意力放在开发团队具有经验和专业能力的领域。

4. 软件的关键性及其非功能性需求。对于一个必须要由外部监管者进行认证的关键性系统而言，可能不得不为该系统创建一个安全或信息安全用况（见第12章）。如果你无法访问软件的源代码，那么这个可能很难做到。如果你的软件有着严格的性能需求，那么使用模型驱动的工程（见第5章）这样的策略可能就不太现实了。模型驱动的工程依赖于从系统的一

个可复用的特定领域模型生成代码。然而，模型驱动的工程中所使用的代码生成器经常会生成相对低效的代码。

方 法	描 述
应用框架	一些抽象类和具体类通过适配和扩展来创建应用系统
应用系统集成	两个或更多的应用系统集成在一起提供扩展的功能
体系结构模式	将支持共性应用系统的标准软件体系结构作为应用开发的基础。第 6、11、17 章中有介绍
面向方面的软件开发	在程序编译时将共享的构件部署到一个应用中的不同地方。在线内容的第 31 章中有介绍
基于构件的软件工程	通过集成符合构件模型标准的构件（由一组对象构成）来开发系统。第 16 章中有介绍
可配置的应用系统	设计特定领域系统使得它们可以按照特定系统客户的需要进行配置
设计模式	出现在多个应用中的通用抽象被表示为设计模式，其中描述了抽象对象和具体对象以及它们的交互。第 7 章中有介绍
ERP 系统	封装了通用业务功能和规则的大型系统面向一个组织进行配置
遗留系统包装	通过定义一组接口并且通过这些接口提供对于遗留系统的访问来对遗留系统（第 9 章）进行“包装”
模型驱动的工程	软件被表示为领域模型和实现无关模型，并且基于这些模型生成代码。第 5 章中有介绍
程序生成器	一个生成器系统中包含了某种类型应用的知识，可以用来在一个用户提供的系统模型基础上生成该领域中的系统
程序库	提供实现了通用抽象的类和函数库，用于复用
面向服务的系统	通过将共享服务链接到一起开发系统，其中的服务可以是外部提供的。在第 18 章中介绍
软件产品线	在一个通用体系结构基础上对某种类型的应用进行泛化，使其可以面向不同客户进行适配
系统之系统	两个或更多的分布式系统被集成起来创建一个新系统。第 20 章中有介绍

图 15-4 支持软件复用的方法

5. 应用领域。在许多应用领域（例如制造和医疗信息系统）中，存在一些通用产品，它们可以通过面向本地环境的配置进行复用。这是最有效的复用方法之一，购买一个系统总是要比重新构建一个新系统要便宜。

6. 系统在其上运行的平台。一些构件模型特定于某个平台，例如 .NET 特定于微软平台。与之相似，通用的应用系统也可能是平台相关的，只能在同样的平台上对它们进行复用。



基于生成器的复用

基于生成器的复用将可复用的概念和知识加入到自动化的工具中，并为工具用户提供一种简单的方式来将特定的代码与这些通用的知识相集成。这一方法通常在特定领域的应用中最有效。领域中已知的问题解决方案被嵌入在生成器系统中，并由用户来选取以生成一个新系统。

可用的复用技术范围很广，以至于在大多数情况下总是存在某种形式的软件复用的可能性。是否实现复用经常是一个管理而非技术问题。管理人员可能会不愿意在需求上让步以使得可复用构件可以使用上。与理解原始开发的风险相比，他们对于与复用相关的风险没有那么好的理解。虽然新软件开发的风险可能更高，一些管理人员还是倾向于面对已知的开发风险而不是未知的复用风险。为了提升企业范围内的复用，可能有必要引入一个关注可复用资产的创建以及有助于复用的过程的复用计划 (Jacobsen, Griss, and Jonsson 1997)。

15.2 应用框架

早期的面向对象开发狂热者们认为，使用面向对象方法的一个关键好处是对象可以在不同的系统中进行复用。然而，经验表明对象经常粒度太小，并且经常针对特定应用进行了特化。与重新实现对象相比，理解和适配一个对象经常要花费更长的时间。现在已经很清楚，在面向对象开发过程中，通过称为框架的更大粒度的抽象来支持面向对象的复用是最好的。

如同其名字所表达的含义，一个框架是一个通用的结构，可以通过扩展来创建更加特定的子系统或应用。Schmidt 等 (Schmidt et al. 2004) 将框架定义为：

一组软件制品 (例如，类、对象、构件) 的有机集合，这些制品相互协作来为一系列相关的应用提供一个可复用的体系结构^①。

框架为很有可能在所有相似类型的应用中使用的通用特性提供支持。例如，一个用户界面框架可以为界面事件处理提供支持，并且提供一组可以用于构造界面显示的小部件，这样开发人员就可以通过为特定的应用增加特定的功能来进行特征化。例如，在一个用户界面框架中，开发人员定义适合于所实现的应用的显示布局。

框架支持设计复用，它们为应用以及系统中特定的类的复用提供了一个骨架体系结构。体系结构通过对象类以及它们的交互来实现。类可以直接复用，也可以使用继承和多态这样的特性对其进行扩展。

框架被实现为某种面向对象编程语言中的一系列具体和抽象对象类。因此，框架是特定于语言的。广泛使用的一些面向对象编程语言都有自己的框架，例如 Java、C#、C++，以及 Ruby、Python 等动态语言。实际上，一个框架还可以包含其他框架，其中每个框架被设计用来支持应用中的一部分开发。可以使用一个框架来创建一个完整的应用或者实现一个应用的一部分，例如图形化用户界面。

使用最广泛的应用框架是 Web 应用框架 (Web Application Framework, WAF)，它们支持动态网站的构造。一个 Web 应用框架的体系结构通常都是基于模型 - 视图 - 控制器 (Model-View-Controller, MVC) 组合模式，如图 15-5 所示。MVC 模式最初是在 20 世纪 80 年代提出的，作为一种图形化用户界面设计方法，允许一个对象存在多种展现方式，并且每种展现都有不同的交互风格。从本质上看，MVC 模式将对象状态从它的展现中分离出去，从而使其状态可以就每一种展现进行更新。

一个 MVC 框架支持以不同的方式展现数据，并且允许与这些展现中的每一个进行交互。当数据通过某种展现方式进行修改之后，系统模型进行修改，与每一个视图相关联的控制器更新它们自己的展现。

① Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." ACM Queue 2(5(July/August)):66-75. doi:10.1145/1016998.1017005.

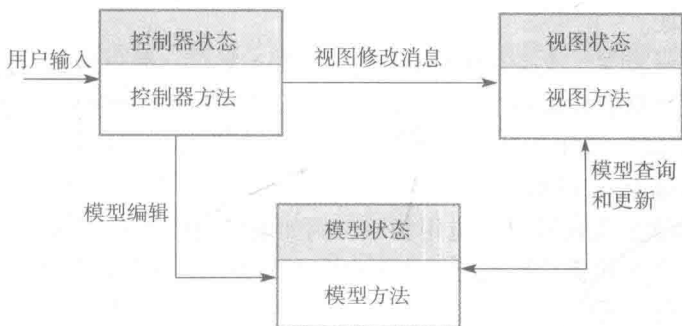


图 15-5 模型 - 视图 - 控制器模式

框架经常是设计模式的实现，如第 7 章中所讨论的。例如，一个 MVC 框架包括观察者（observer）模式、策略（strategy）模式、组合模式（composite）以及 Gamma 等人（Gamma et al. 1995）的书中所讨论的其他一些模式。模式的通用特性以及它们所使用的抽象类和具体类使得可扩展性成为可能。没有模式，框架几乎肯定会变得没有实用价值。

虽然框架所包含的功能略有不同，Web 应用框架所提供的构件和类通常都支持以下这些功能。

1. 信息安全。Web 应用框架可以包含一些类来帮助实现用户身份认证（登录）和访问控制，以确保用户在系统中只能访问被允许的一些功能。
2. 动态 Web 页。提供一些类来帮助你定义 Web 页模板，以及动态地填入来自系统数据库中的特定数据。
3. 数据库集成。框架自身通常并不包含数据库，而是假设会使用一个独立的数据库，例如 MySQL。框架可以包含一些类来提供对于不同数据库的抽象接口。
4. 会话管理。Web 应用框架通常都会包含一些类来创建并管理会话（用户与系统之间的一系列交互）。
5. 用户交互。Web 应用框架提供了 AJAX（Holdener 2008）和 HTML5 支持（Sarris 2013），它们允许创建交互式的 Web 页。它们可能会包含一些类来允许创建与设备无关的界面，这些界面可以自动地适应移动电话和平板电脑。

为了使用一个框架实现一个系统，要增加一些继承自框架中的抽象类的操作的具体类。此外，还会定义回调（callback），即作为对框架所识别的事件的应答而被调用的方法。框架对象，而非应用特定的对象，负责系统中的控制。Schmidt 等人（Schmidt, Gokhale, and Natarajan 2004）将此称为控制转置（inversion of control）。

作为对来自用户界面和数据库的事件的应答，框架对象调用那些链接到用户提供的功能的“钩子方法”。用户提供的功能定义了应用应当如何响应事件（见图 15-6）。例如，一个框架有一个方法来处理来自环境的鼠标点击。这个方法被称为钩子方法，必须配置这个方法来调用合适的应用方法来处理鼠标点击。

Fayad 和 Schmidt（Fayad and Schmidt 1997）

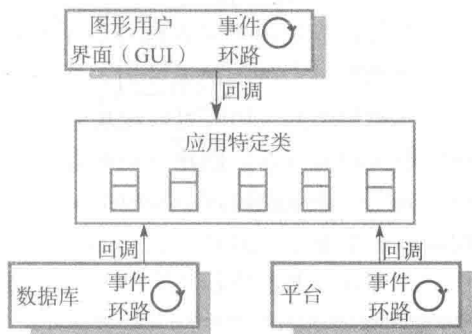


图 15-6 框架中的控制转置

讨论了其他 3 种框架。

1. 系统基础设施框架, 支持系统基础设施 (例如, 通信、用户界面、编译器) 的开发。

2. 中间件集成框架, 包含一组支持构件通信和信息交换的标准以及相关的对象类。这类框架的例子包括微软的 .NET 以及 EJB (Enterprise Java Beans)。这些框架对标准化的构件模型提供了支持, 如第 16 章所述。

3. 企业应用框架, 关注特定的应用领域, 例如电信或财务系统 (Baumer et al. 1997)。这些框架包含应用领域知识并且支持最终用户应用的开发。这些框架现在并没有广泛使用, 并且已经被软件产品线所取代^①。

使用框架构造的应用可以通过软件产品线或者应用族的思想成为进一步复用的基础。因为这些应用都是用一个框架构造的, 修改产品族成员创建其他系统实例经常是一个很直观的过程。其中包括重写已加入框架的具体类和方法。

框架是一种非常有效的复用方法。然而, 将它们引入到软件开发过程中是很昂贵的, 因为它们内在的复杂性, 可能要花上几个月的时间去学习如何使用框架。评估可用的框架, 然后挑选一个最合适的, 这样做会很困难和昂贵。调试基于框架的应用比调试原始代码更困难, 因为你可能不理解框架方法是如何交互的。调试工具可能会提供关于被复用的框架构件的信息, 但是开发人员不理解这些信息。

15.3 软件产品线

当一家企业不得不支持许多相似但不完全相同的系统时, 其中最有效的一种复用方法是创建一个软件产品线。硬件控制系统经常使用这种方法进行开发, 就像物流或医疗系统这样的领域中的特定领域应用。例如, 一个打印机制造商要开发打印机控制软件, 其中针对每一种型号的打印机都有一个特定的版本。这些软件版本有很多共性的部分, 合理的做法是, 创建一个核心产品 (产品线), 然后针对每种打印机型号进行适配是合理的。

一个软件产品线是具有一个共性体系结构以及共享构件的一组应用, 其中每一个应用都进行了特征化以反映特定的客户需求。核心系统的设计使得可以对其进行配置和适配以适应不同客户或装备的需要。这可能会包括配置一些构件、实现一些额外的构件、修改一些构件, 从而反映新需求。

通过适配一个应用的通用版本来开发应用, 意味着应用代码中很大一部分都可以在每个系统中进行复用。测试也可以简化, 因为对应用中很大一部分的测试都可以被复用, 因此可以降低总体的应用开发时间。工程师通过软件产品线了解、学习应用领域, 进而可以成为能够快速开发出新应用的专家。

软件产品线通常是在已有应用的基础上涌现出来的。也就是说一个组织开发一个应用, 然后当需要一个相似的系统时, 采用非正式的方法在新应用中复用来自前一个应用的代码。开发其他相似的应用时可以使用同样的过程。然而, 修改会破坏应用结构, 因此随着更多的新应用实例的开发, 创建一个新版本将变得越来越困难。其结果是, 接下来可能会做出设计一个通用产品线的决定。这包括识别产品实例中的共性功能并且开发一个基准应用, 这个应用接着将用于未来的开发。

设计这个基准应用 (见图 15-7) 的目的是简化复用和重配置。一般而言, 一个基准应用

① Fayad, M. E., and D. C. Schmidt. 1997. "Object-Oriented Application Frameworks." *Comm. ACM* 40(10): 32-38. doi:10.1145/262793.262798.

包括以下这些内容。

- 1. 提供基础设施支持的核心构件。这些构件在开发新的产品实例时通常不会进行修改。
- 2. 可以进行修改和配置从而面向一个新应用对它们进行特征化的可配置构件。有时候可以使用内建的构件配置语言对这些构件进行重新配置而不需要修改它们的代码。
- 3. 一些特征化的、领域特定的构件，其中的一些或全部在创建产品线的新实例时会被替换掉。

应用框架和软件产品线有很多共性。它们都支持一个共性的体系结构和一些共性构件，并需要新的开发来创建一个系统的特定版本。这些方法的区别主要在于以下几个方面。

- 1. 应用框架依赖于面向对象特性（例如，继承和多态）来实现对框架的扩展。一般而言，框架代码没有修改，可能的修改被限制在框架所支持的那些地方。软件产品线并不一定要使用面向对象方法进行创建。可以对应用构件进行替换、删除或重写。至少在原则上对于可以做的修改是没有限制的。
- 2. 大部分应用框架提供了泛化的支持而不是特定领域的支持。例如，有支持创建基于 Web 的应用的应用框架。软件产品线通常包含详细的领域和平台信息。例如，可以有一个关注基于 Web 的健康记录管理应用的软件产品线。
- 3. 软件产品线经常是设备的控制应用。例如，可以有针对一个打印机产品族的软件产品线。这意味着产品线必须为硬件接口提供支持。应用框架通常是面向软件的，它们通常并不会包含硬件交互构件。
- 4. 软件产品线由一系列相关的应用族构成，属于同一个组织。当创建新的应用时，起点经常是应用族中最接近的产品成员，而不是通用的核心应用。

如果要使用一种面向对象编程语言开发一个软件产品线，那么可以使用一个应用框架作为系统基础。通过在框架基础上使用它的内建机制扩展特定领域的构件来创建产品线核心。然后，在第二个开发阶段中为不同的客户创建不同的系统版本。例如，可以使用一个基于 Web 的框架来构建一个基于 Web 的咨询台软件产品线的核心。这个“咨询台产品线”接着可以被进一步特征化以提供特定类型的咨询台支持。

一个软件产品线的体系结构经常反映了一种通用的应用特定的体系结构风格或模式。例如，考虑一个设计用于应急服务的车辆调度的产品线系统。这个系统的操作人员接听事故来电，寻找合适的车辆来响应事故处理，然后调度车辆到事故地点。这样一个系统的开发者会面向警察、消防和救护车服务进行产品销售。

这个车辆调度系统是一个通用的资源分配和管理系统（见图 15-8）的一个例子。资源管理系统使用一个可用资源的数据库，并且包含实现使用该系统的企业所确定的资源分配策略的构件。用户与资源管理系统交互以请求和释放资源，并且询问关于资源及其

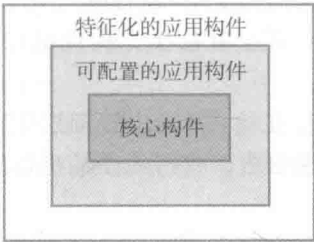


图 15-7 面向一个产品线的基准系统组织



图 15-8 一个资源管理系统的体系结构

可用性的问题。

你可以在图 15-9 中看到这个四层结构可以如何进行实例化，其中显示了可以被包含在一个车辆调度系统产品线中的模块。这个产品线系统中每一层上的构件如下：

1. 在交互层上，构件提供了一个操作人员显示界面以及与所使用的通信系统之间的接口；
2. 在输入/输出管理层（第 2 层）上，构件处理操作人员身份认证，生成事故以及调度车辆报告，支持地图输出以及路径规划，并且为操作人员提供了一种查询系统数据库的机制。
3. 在资源管理层（第 3 层）上，构件允许对车辆进行定位和调度，更新车辆和装备状态，并且对事故的细节进行日志记录。
4. 在数据库管理层上，除了通常的事务管理支持，还有分开的车辆、装备、地图数据库。



图 15-9 一个车辆调度系统的产品线体系结构

为了创建这个系统的一个新实例，可能要修改各个构件。例如，警察有大量的车辆但车辆类型相对较少。与之相比，消防服务有很多不同类型的专业车辆但车辆数量相对较少。因此，当你为这些服务实现一个系统时，可能要定义不同车辆数据库结构。

应用开发时可能会进行以下这些不同类型的软件产品线特化开发。

1. 平台特化。应用面向不同的平台开发相应的版本。例如，一个应用可能存在针对 Windows、Mac OS、Linux 平台的不同版本。在这种情况下，应用的功能通常没有变化；只有那些与硬件和操作系统接口的构件会被修改。

2. 环境特化。应用创建不同的版本来处理不同的运行环境和外围设备。例如，一个应急服务系统可能存在不同的版本，根据每个服务所使用的通信硬件。例如，警察无线电所内置的加密机制必须要使用。产品线构件要进行修改以反映所使用的设备的功能和特点。

3. 功能特化。应用面向有着不同需求的特定客户创建不同的版本。例如，一个图书馆自动化系统可能会根据该系统是用于公共图书馆、参考书阅览室还是大学图书馆来进行修改。

这种情况下,实现功能的构件可能要修改,而新的构件可能会加入到系统中。

4. 过程特化。对系统进行调整以应对不同的特定业务过程。例如,一个订购系统可能会进行调整以应对某一个公司的集中式订购过程以及另一个公司的分布式过程。

图 15-10 描述了扩展一个软件产品线以创建一个新应用的过程。该过程中所包含的活动如下。

1. 抽取利益相关者需求。可以以一个常规的需求工程过程作为开始。然而,由于已经存在一个系统,因此可以展示该系统并且让利益相关者试验该系统,表达他们的需求以便对已提供的功能进行修改。

2. 选择一个最接近需求的现有系统。当创建一个新的产品线成员时,可以首先选择一个最接近的产品实例。分析了需求之后,选择最接近的产品线成员进行修改。

3. 重新协商需求。随着所要求的修改的更多细节出现以及对项目进行计划,一些需求可能需要与客户重新进行协商以尽量减少对基准应用所做的修改。

4. 适配修改已有系统。向已有的系统中开发并加入新模块,同时对已有的系统模块进行适配修改以满足新需求。

5. 交付新的产品族成员。将产品线的新实例交付给客户。可能需要一些部署时的配置来反映系统将被使用的特定环境。在这个阶段,你应该记录产品的关键特征,从而使得该产品可以在未来作为其他系统开发的基础。

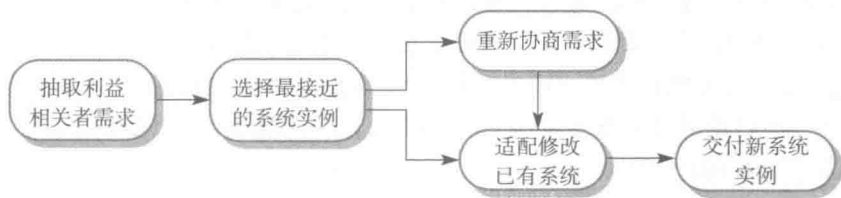


图 15-10 产品实例开发

当创建一个产品线的新成员时,可能要在尽量复用通用应用以及满足利益相关者的详细需求之间进行权衡。系统需求越详细,已有构件满足这些需求的可能性越低。然而,如果利益相关者愿意灵活一点,限制所需要的系统修改,那么通常可以更快交付系统并且成本更低。

软件产品线被设计成具有重配置的能力。这种重配置可能包括增加或移除系统构件、为系统构件定义参数和约束、加入业务过程知识。这一配置可以发生在开发过程的不同阶段,具体如下。

1. 设计时配置。开发软件的组织机构开发、选取或适配修改构件来为一个客户创建一个新系统,通过这种方式修改一个通用的产品线核心。

2. 部署时配置。设计一个通用系统,客户或与客户一起工作的咨询人员可以对其进行配置。关于客户特定需求以及系统运行环境的知识包含在通用系统所使用的配置数据中。

当一个系统在设计时进行配置时,供应商在一个通用系统或者一个已有的产品实例基础上开始开发。供应商通过修改和扩展系统中的模块来创建一个提供所需要的客户功能的特定系统。这通常包括修改和扩展系统的源代码,从而获得比部署时配置更大的灵活性。

当无法使用一个系统中已有的部署时配置手段来开发一个系统版本时,可以使用设计时配置。然而,随着时间的流逝,当你已经创建了多个具有相似功能的产品族成员时,你可能

会决定对核心产品线进行重构以包含已经在多个应用产品族成员中实现的功能。这样就使得这些新功能可以在系统部署时进行配置了。

部署时配置包括使用一个配置工具来创建一个特定的系统配置，这些配置记录在一个配置数据库或者一组配置文件中（见图 15-11）。运行中的系统（可能运行在一台服务器上或者作为独立的系统运行在个人电脑上）在运行中读取这个数据库，从而使系统的功能可以根据运行上下文进行特化。

一个系统中可能提供的部署时配置包括如下这些不同层次。

1. 构件选取，即选取系统中提供了所需要功能的模块。例如，在一个病人信息系统中，选择一个允许将医疗影像（X 光、CT 扫描等）链接到病人医疗记录的影像管理构件。

2. 工作流和规则定义，定义应当适用于用户所输入的或者系统所生成的信息的工作流（信息如何按照一个个阶段进行处理）验证规则。

3. 参数定义，指定特定的系统参数的值以反映正在创建的应用实例的需求。例如，可以指定允许用户输入的数据字段的最大长度或者附加到系统上的硬件的特性。

对于大型系统而言，部署时配置可以很复杂，可能要花费几个月来为一个客户配置并测试一个系统。大型的可配置系统可以通过提供软件工具（例如规划工具）来支持配置过程。15.4.1 节将进一步讨论部署时配置。这一讨论覆盖了必须进行配置以便在不同的运行环境中工作的应用系统的复用。

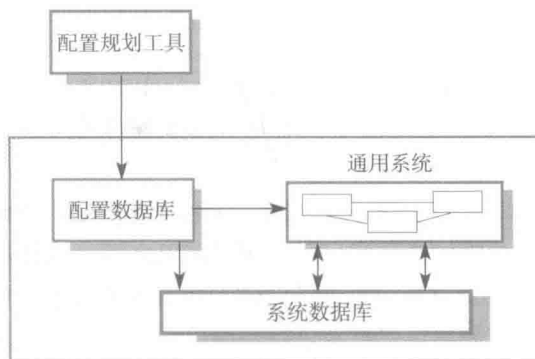


图 15-11 部署时配置

15.4 应用系统复用

一个应用系统产品是一个可以在不修改系统源代码的情况下通过适配来满足不同客户需要的软件系统。应用系统是由一个系统供应商面向通用市场开发的，而不是特别为单个客户开发的。这些系统产品有时候被称为商用第三方系统（Commercial Off-the Shelf System, COTS）产品。然而，“商用第三方系统”这个术语基本都是在军事系统中使用的，因此本书倾向于将这些系统产品称为“应用系统”。

事实上所有的桌面业务软件以及许多基于服务器的系统都是应用系统。这种软件被设计用于通用目的，其中包括了很多特征和功能。因此，它们具有在不同环境中复用以及作为不同应用的一部分的潜力。Torchiano 和 Morisio（Torchiano and Morisio 2004）也发现开源产品经常在不进行修改以及不看源代码的情况下使用。

应用系统产品通过使用内置的配置机制来进行适配，这些机制允许根据特定的客户需要对系统的功能进行裁剪。例如，一个医院病人记录系统为不同类型的病人定义了不同的输入表格和输出报告。其他一些配置特征可能会允许系统接受扩展功能，或者检查用户输入以确保其合法性的插件。

自 20 世纪 90 年代后期开始，大型企业已经广泛采用了这一软件复用方法，因为与定制化的软件开发相比，这种方法提供了如下这些显著的收益。

1. 与其他类型的复用相比，可以更快地部署一个可靠的系统。

2. 有可能看到应用提供了什么功能, 因此可以更容易地判断这些功能是否合适。其他企业可能已经使用了这些应用, 因此有一些该系统的经验。

3. 通过使用已有的软件来避免一些开发风险。然而, 这种方法也有它自己的风险, 下面会讨论。

4. 业务部门可以关注他们自己的核心活动, 而不用在信息技术 (IT) 系统开发上花费大量的资源。

5. 随着运行平台的演化, 技术更新可以得到简化, 因为这些是应用系统供应商的责任而不是客户的责任。

当然, 这种软件工程方法也有如下这些问题。

1. 通常要对需求进行调整以反映第三方应用系统的功能以及运行模式。这可能会给已有的业务过程带来干扰性的变化。

2. 应用系统可能会基于一些现实中无法改变的假设。因此, 客户必须调整自己的业务来反映这些假设。

3. 为一个企业选择正确的应用系统可能是一个困难的过程, 特别是当许多这类系统并没有很好的文档描述的时候。做出错误的选择意味着可能无法让新系统按照所要求的方式工作。

4. 可能会缺少支持系统开发的本地专家。其结果是, 客户不得不依赖于供应商和外部的咨询人员来给出开发建议。这些建议可能会倾向于销售他们的产品和服务, 而没有花费足够的时间来理解客户的真实需要。

5. 系统供应商控制系统的支持和演化。这一过程中可能会做出一些导致客户困难的修改。

应用系统可以作为独立的系统使用, 或者结合起来使用 (其中两个或多个系统集成在一起)。独立的系统由一个通用的应用组成, 来自于单个供应商并且按照客户需求进行了配置。集成的系统是指通过对来自各个系统 (经常来自不同的供应商) 的功能进行集成来创建一个新的应用系统。图 15-12 总结了这些方法的区别。15.4.2 节将会讨论应用系统集成。

可配置的应用系统	应用系统集成
单个产品提供一个客户所需要的功能	多个不同的应用系统集成到一起以提供定制化的功能
基于一个通用的解决方案以及标准化的过程	可以针对客户过程开发灵活的解决方案
开发关注系统配置	开发关注系统集成
系统供应商负责维护	系统所有者负责维护
系统供应商为系统提供平台	系统所有者为系统提供平台

图 15-12 独立的应用系统和集成的应用系统

15.4.1 可配置的应用系统

可配置的应用系统是通用应用系统, 用于支持特定的业务类型、业务活动或者一个完整的业务企业。例如, 一个面向牙科医生的系统可以处理预约、提醒、牙齿记录、病人召回和账务处理。在更大的范围上, 一个企业资源规划 (Enterprise Resource Planning, ERP) 系统可以支持一个大型企业中的生产、订单、客户关系管理过程。

特定领域的应用系统，例如支持业务功能的系统（例如文档管理），提供了很多潜在用户都可能会需要的功能。然而，它们也包含了一些关于用户如何工作的内在假设，而这些假设可能会在特定情况下导致问题。例如，一个支持大学中的学生注册的系统可能会假设学生只会在一所大学中注册一个学位。然而，如果多所大学相互合作提供联合学位，那么在实践中有可能无法在系统中表示这一细节。

ERP 系统，例如 SAP 和 Oracle 所开发的那些系统，是大规模集成系统，被设计用于支持各种业务实践，例如订货和开票、库存管理、制造规划（Monk and Wagner 2013）。这些系统的配置过程会收集关于客户的业务和业务过程的详细信息，并将这些信息保存在配置数据库中。这经常会要求掌握有关配置表示法和工具的详细知识，并且通常由与系统客户一起工作的咨询人员进行。

一个通用 ERP 系统包括一些模块，这些模块可以用不同的方式组合从而为客户创建一个系统。配置过程包括选择包含哪些模块、配置各个模块、定义业务过程和业务规则、定义系统数据库的结构和组织。一个支持一系列业务功能的 ERP 系统的整体体系结构模型如图 15-13 所示。

这个体系结构的关键特性如下。

1. 若干支持不同业务功能的模块。这些是大粒度的模块，支持企业的整个部门。在图 15-13 所示的例子中，系统中的模块包括：一个支持采购的模块、一个支持供应链管理的模块、一个支持商品配送的物流模块、一个维护客户信息的客户关系管理模块。
2. 与每个模块相关联的一组定义好的业务过程模型，它们与模块中的活动相关。例如，订购过程模型可以定义如何创建并审批订单。这其中会指定下订单过程中所涉及的角色和活动。
3. 维护所有相关业务功能的信息的通用数据库。因此，没有必要在不同部分的业务中复制相关信息，例如客户详细信息。
4. 适用于数据库中所有数据的一组业务规则。因此，当数据从某个功能输入到系统中时，这些规则应当确保这些数据与其他功能所需要的数据相一致。例如，一个业务规则可能会要求所有的支出报销都必须由比报销人级别更高的人批准。

ERP 系统在几乎所有大型企业中都使用以支持他们的部分或所有功能。因此，它们是一种非常广泛使用的软件复用形式。这种复用方法一个很明显的限制是，客户应用的功能受到 ERP 系统的内置模块的功能的限制。如果一个企业需要额外的功能，那么可能只有开发一个补充系统来提供这个功能。

此外，客户企业的过程和运行必须在 ERP 系统的配置语言中进行定义。这种语言蕴含着系统供应商所理解的业务过程，而这些假设以及客户业务中所使用的概念和过程之间可能存在不匹配。客户业务模型和 ERP 系统所使用的系统模型之间的严重不匹配使得 ERP 系统很有可能会无法满足客户的真正需求（Scott 1999）。

例如，在一个销售给一所大学的 ERP 系统中，一个基本的系统概念是客户。在这个系统中，一个客户是从一个供应商那里购买商品和服务的外部代理。这一概念导致配置该系统

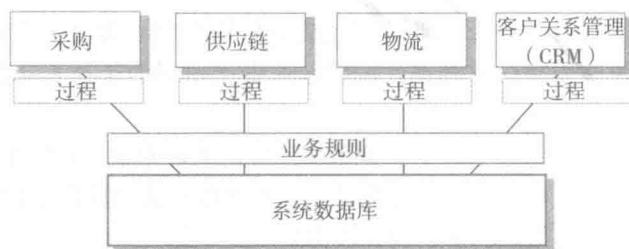


图 15-13 一个 ERP 系统的体系结构

时的巨大困难。大学实际上没有客户。他们与一系列人和组织有类似于客户的关系,例如学生、研究资助机构、教育机构等。这些关系都跟客户关系(其中一个人或企业从另一个人或企业那里购买商品或服务)不相符。在这个特定案例中,他们花费了好几个月的时间来解决然这个不匹配,然而最终的解决方案也只能部分满足大学的需求。

ERP系统通常需要进行广泛的配置来使它们适应安装系统的每个组织的需求。这种配置可能包括:

1. 从系统中选取所需要的功能,例如通过确定系统应该包括哪些模块;
2. 建立一个数据模型,从而定义组织的数据在系统数据库中应该用什么样的结构;
3. 定义适用于这些数据的业务规则;
4. 定义所期望的与外部系统的交互;
5. 设计输入表单以及系统所生成的输出报告;
6. 设计符合系统所支持的过程模型的新业务过程;
7. 设置参数,从而定义系统如何在所依赖的平台上进行部署。

配置设置完成后,新的系统就可以进行测试了。当系统是通过配置而不是使用一种传统语言编程得到的时候,测试是一个大问题。以下是两个主要原因。

1. 测试自动化很难或者根本不可能。可能没什么办法让测试框架(例如JUnit)访问相关的API,因此系统必须由测试人员向系统输入数据来进行手工测试。此外,系统经常是以非正式的方式描述的,因此在没有最终用户的大力帮助下定义测试用例很困难。

2. 系统错误经常很微妙并且特定于业务过程。应用系统或ERP系统是可靠的平台,因此技术系统失效很少发生。所发生的问题经常是由于配置系统的人和用户利益相关者之间的误解导致的。对最终用户过程的详细信息不了解的系统测试人员无法发现这些错误。

15.4.2 集成的应用系统

集成的应用系统包括两个或更多的应用系统,或者有时候,包括遗产系统。当没有单个的应用系统满足所有的需要,或者当试图将一个新应用系统与已经在使用的系统相集成时,可以用这种方法。如果有定义好的API或者服务接口,那么成员系统可以通过这些API或者服务接口进行交互。或者,它们也可以通过将一个系统的输出与另一个系统的输入连接起来,或者更新应用所使用的数据库,以便将这些成员系统组装起来。

为了开发集成的应用系统,你要做出以下这些设计选择。

1. 哪一个应用系统提供了最合适的功能?典型情况下会有多个可用的系统产品,它们可以通过不同的方式进行结合。如果你对 these 应用系统都没有什么经验,那么确定哪个产品最合适可能很难。

2. 数据将如何交换?不同的系统通常使用独特的数据结构和格式,必须编写适配器来将一种表示转换为另一种。这些适配器是与作为系统成员的应用系统一起运行的运行时系统。

3. 一个产品的哪些特征会被实际用到?各个应用系统所包括的功能可能比所需要的多,而不同产品之间的功能可能存在重复。必须确定哪个产品中的哪些特征最适合于你的需求。如果有可能,还应当拒绝对未使用功能的访问,因为这可能会干扰正常的系统运行。

考虑下列场景中的应用系统集成问题。一个大型组织想要开发一个允许员工在自己的工作台上上下订单的采购系统。通过在组织范围内引入这个系统,该企业预计每年可以节省500万美元。通过集中式的购买,新的采购系统可以保证总是从提供最优价格的供应商那里进行

订购,并且可以减少与此相关的管理成本。与手工的系统一样,该系统也包括从一个供应商那里选择商品、创建订单、审批订单、发送订单给供应商、接收商品、确认应该支付等部分。

该企业有一个遗留的由集中采购办公室使用的订购系统。这个订单处理软件与一个已有的开票和配送系统相集成。为了创建新的订购系统,遗留系统与一个基于 Web 的电子商务平台以及一个处理用户沟通的电子邮件系统进行集成。最终的采购系统结构如图 15-14 所示。

这个采购系统应当是一个客户-服务器系统,其中客户端会使用标准的 Web 浏览器和电子邮件系统。在服务器上,电子商务平台必须通过适配器与已有的订购系统相集成。电子商务系统有自己的订单、配送确认等格式,这些都必须转换为订购系统所使用的格式。电子商务系统使用电子邮件系统来向用户发送通知,但是订购系统从来没有考虑这个目的。因此,必须开发另一个适配器来将订购系统的通知转换为邮件消息。

通过集成已有的应用系统可以节省数月,有时甚至是数年的实现工作量,而且开发和部署一个系统的时间可以大幅度减少。上面所描述的这个采购系统在一个很大的企业里面用了 9 个月完成了实现和部署。而最初预计要用 3 年的时间用 Java 开发一个可以与遗留的订购系统相集成的订购系统。

如果使用面向服务的方法,那么应用系统集成还可以简化。本质上看,一个面向服务的方法意味着允许通过标准的服务接口访问应用系统,而每个独立的功能单元都有一个服务。一些应用会提供服务接口,但有时候这个服务接口必须由系统集成者来实现。从本质上看,你必须开发一个包装器来隐藏应用并且提供外部可见的服务(见图 15-15)。这种方法对于必须与新应用系统相集成的遗留系统尤其有价值。

原则上,集成应用系统与集成任何其他构件都是一样的:必须理解系统接口并使用它们与软件通信;必须在特定的需求以及快速的开发和复用之间进行权衡;必须设计一个允许应用系统之间互操作的系统体系结构。

然而,这些产品自身通常都是大型系统,它们经常作为独立的系统进行销售,这便引入了一些额外的问题。Boehm 和 Abts (Boehm and Abts 1999) 强调了下面这 4 个重要的系统集成问题。

1. 对功能和性能缺少控制。虽然一个产品所发布的接口可能会提供所需要的手段,但是系统可能没有适当地实现或者可能会运行得很糟糕。产品可能存在一些隐藏的操作会干扰它在特定情形中的使用。修复这些问题对于系统集成者可能很重要,但是对于产品供应商可能

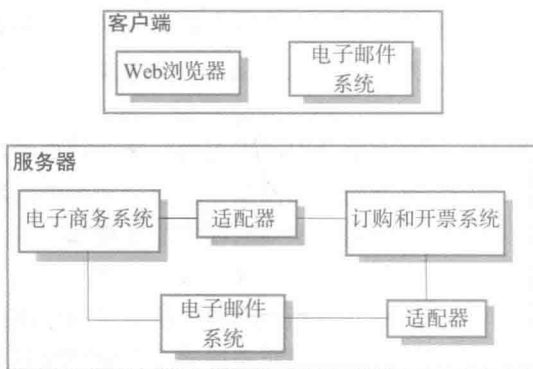


图 15-14 一个集成的采购系统

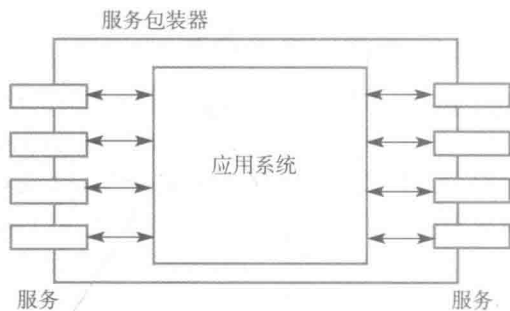


图 15-15 应用包装

并不是一个关注点。如果想复用应用系统的话,用户可能不得不为这些问题寻找变通方案。

2. 系统互操作性的问题。让各个应用系统一起工作有时候很难,因为每个系统都包含一些关于自己将被如何使用的假设。Garlan 等人 (Garlan, Allen, and Ockerbloom 1995) 描述了他们集成 4 个应用系统的经验,他们发现其中 3 个产品是基于事件的,但是使用了不同的事件模型。每个系统都假设它们对于事件序列拥有排他的访问。其结果是,集成非常困难。项目需要花费的时间是最初估计的 5 倍。项目进度延长到了 2 年,而不是原来估计的 6 个月。

在 10 年后的回顾性分析中, Garlan 等人 (Garlan, Allen, and Ockerbloom 2009) 得出的结论是,他们所发现的集成问题还没有解决。Torchiano 和 Morisio (Torchiano and Morisio 2004) 发现许多应用系统中缺乏对标准的遵循,这意味着集成会比预计的更困难。

3. 无法控制系统演化。应用系统的供应商自己决定为应对市场压力系统该如何变更。特别是对于个人电脑产品,新的版本经常会频繁发布而且可能会不与此前的版本兼容。新的版本可能会包含一些额外的不需要的功能,而此前的版本可能会不可用或者不再支持了。

4. 来自系统供应商的支持。来自系统供应商的可用的支持水平差别很大。当开发者在没有系统的源代码以及详细文档的情况下出现问题时,供应商支持尤其重要。虽然供应商会承诺提供服务,变化的市场和经济环境会使得他们很难履行这一承诺。例如,一个系统供应商可能会因为需求量太少而决定不再继续某个产品,或者被另一个不愿意为已经销售的产品提供支持的公司所取代。

Boehm 和 Abts 估计,在很多情况下集成化应用系统的维护演化成本可能会更高。上面这些困难都是生命周期问题;它们并不仅仅影响系统最初的开发。参与系统维护的人中来自最初的系统开发者越少,集成系统越容易出现问題。

要点

- 有很多不同的软件复用方式,从库中的类和方法的复用到整个应用系统的复用。
- 软件复用的优势包括更低的成本、更快的软件开发、更低的风险,系统可依赖性也会提高。可以通过将专家的专业知识集中在可复用构件的设计上,使专家的工作更有效。
- 应用框架由一系列具体和抽象对象组成,它们通过特化以及增加新的对象来进行复用。它们通常都会通过设计模式来将好的设计实践包含其中。
- 软件产品线是从一个或多个基准应用基础上开发出来的相关的应用。对一个通用系统进行适配和特化来满足特定的功能、目标平台或者运行配置需求。
- 应用系统复用关注复用大规模的成品系统。这些系统提供了很多功能,它们的复用可以大幅度减少成本和开发时间。系统可以通过配置单个的通用应用系统或者集成两个或多个的应用系统进行开发。
- 应用系统复用的潜在问题包括缺少对功能、性能和系统演化的控制;需要来自外部供应商的支持;确保系统互操作方面的困难。

阅读推荐

《Overlooked Aspects of COTS-Based Development》是一篇有趣的文章,对使用基于第三方成品软件的方法的开发人员进行了调研,并对他们遇到的问题进行了讨论。(M.

Torchiano and M. Morisio, IEEE Software, 21 (2), March-April 2004) <http://dx.doi.org/10.1109/MS.2004.1270770>

《CRUISE—Component Reuse in Software Engineering》这本电子书介绍了很多与复用相关的话题,包括案例研究、基于构件的复用、复用过程。然而,其中的应用系统复用比较少。(L. Nascimento et al., 2007) http://www.academia.edu/179616/CRUISE_-_Component_Reuse_in_Software_Engineering

《Construction by Configuration: A New Challenge for Software Engineering》这篇受邀论文中,讨论了通过配置已有系统构造新应用的问题和困难。(I. Sommerville, Proc. 19th Australian Software Engineering Conference, 2008) <http://dx.doi.org/10.1109/ASWEC.2008.75>

《Architectural Mismatch: Why Reuse Is Still So Hard》这篇文章回顾了以前的一篇讨论复用和集成应用系统的论文中所讨论的问题。作者总结出,虽然取得了一些进展,但是在各个系统的设计者所做的相互冲突的假设中仍然存在问题。(D. Garlan et al., IEEE Software, 26(4), July-August 2009) <http://dx.doi.org/10.1109/MS.2009.86>

网站

本章的PPT: <http://software-engineering-book.com/slides/chap15/>

支持视频的链接: <http://software-engineering-book.com/videos/software-reuse/>

练习

- 15.1 阻碍软件复用的主要技术因素和非技术因素分别是什么?你个人是否复用了大量软件,如果没有的话,为什么?
- 15.2 为什么来自复用已有软件的成本节省并不是简单地与所复用的构件的规模成正比?
- 15.3 说出4种你会建议不要进行复用的情形。
- 15.4 解释应用框架中的“控制转置”是什么意思。如果你集成了两个独立的最初都是使用同样的应用框架创建的系统,为什么这一方法可能会导致问题?
- 15.5 根据第1章和第7章中所介绍的气象站系统的例子,为一个关注远程监控和数据收集的应用族提出一个产品线体系结构。你应该将你的体系结构呈现为一种分层模型,显示每一层上可能包含的构件。
- 15.6 大多数桌面软件,例如字处理软件,都可以用一些不同的方式进行配置。检查你经常会使用的软件,列出这些软件的配置选项。说出一些用户在配置这些软件时可能会碰到的困难。微软的Office(或者其他类似的开源软件)是一个很好的例子。
- 15.7 为什么许多大型企业都选择ERP系统作为新系统的基础?在一个组织中部署一个大规模ERP系统会出现什么问题?
- 15.8 说出使用现有的应用系统来构造系统时可能会出现6个可能的风险。企业可以采取哪些步骤来降低这些风险?
- 15.9 为什么在通过集成应用系统来构造系统时通常会需要适配器?说出在实践中编写适配器软件来链接两个应用系统时可能会出现3个问题。
- 15.10 软件复用导致了一些版权和知识产权问题。如果一个客户付钱让一个软件合同商来开发一个系统,谁有权复用所开发的代码?软件合同商有权将这些代码作为一个通用构件的基础来使用吗?什么样的付款机制可以用来补偿可复用构件的提供者?讨

论这些问题以及其他与软件复用相关的伦理道德问题。

参考文献

- Baumer, D., G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zullighoven. 1997. "Framework Development for Large Systems." *Comm. ACM* 40 (10): 52–59. doi:10.1145/262793.262804.
- Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.
- Fayad, M.E., and D.C. Schmidt. 1997. "Object-Oriented Application Frameworks." *Comm. ACM* 40 (10): 32–38. doi:10.1145/262793.262798.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Garlan, D., R. Allen, and J. Ockerbloom. 1995. "Architectural Mismatch: Why Reuse Is So Hard." *IEEE Software* 12 (6): 17–26. doi:10.1109/52.469757.
- . 2009. "Architectural Mismatch: Why Reuse Is Still so Hard." *IEEE Software* 26 (4): 66–69. doi:10.1109/MS.2009.86.
- Holdener, A.T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.
- Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.
- Monk, E., and B. Wagner. 2013. *Concepts in Enterprise Resource Planning, 4th ed.* Independence, KY: CENGAGE Learning.
- Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.
- Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." *ACM Queue* 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.
- Scott, J. E. 1999. "The FoxMeyer Drug's Bankruptcy: Was It a Failure of ERP." In *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*. Milwaukee, WI. <http://www.uta.edu/faculty/weltman/OPMA5364TW/FoxMeyer.pdf>
- Torchiano, M., and M. Morisio. 2004. "Overlooked Aspects of COTS-Based Development." *IEEE Software* 21 (2): 88–93. doi:10.1109/MS.2004.1270770.

基于构件的软件工程

目标

本章的目标是阐述一种基于可复用的、标准化构件组装的软件复用方法。阅读完本章后，你将：

- 理解可以包含在程序中作为可执行元素的软件构件的含义；
- 了解软件构件模型的关键要素以及中间件为这些模型提供的支持；
- 了解在基于构件的软件工程（Component-Based Software Engineering, CBSE）过程中复用的关键活动和关于复用的 CBSE 过程；
- 了解 3 种不同类型的构件组装以及当组装构件以创建新构件或系统时必须解决的一些问题。

基于构件的软件工程作为一种基于复用软件构件的软件系统开发方法，是在 20 世纪 90 年代末期出现的。它的产生是由于设计者们在使用面向对象的开发过程中所受到的挫折，这种挫折缘于面向对象开发不能够像人们最初所期待的那样完成广泛的复用。单个对象类有太多细节且具有特定性，通常需要在编译时与应用绑定。我们必须拥有对类的详细知识来应用它们。一般来讲这意味着不得不掌握构件的源代码。这也意味着，销售和发布对象作为单独的可复用的构件实际上是不可行的。

构件是比对象更高层次的抽象，是由它们的接口来定义的。它们一般比对象大，所有的实现细节对其他构件是隐藏的。CBSE 是定义、实现、集成或组装松散耦合的独立构件成为系统的过程。

对于大型商业系统而言，CBSE 已经成为重要的软件开发方法，因为软件系统变得更大、更复杂，且用户要求开发更可靠、发布和部署更快的软件。我们处理这种复杂性并更快交付软件的方法是复用软件构件，而不是重新实现软件构件。

基于构件的软件工程的要素有：

1. 完全由接口进行规格说明的独立构件。构件接口与构件实现之间应该明确地分离开来，这意味着当用另一种方法实现构件代替其他的实现时系统不发生任何改变。
2. 构件标准使构件集成变得更为容易。这些标准包含在构件模型之中，在最低程度上规定了构件接口应该如何定义，如何实现构件间的交互。一些模型定义了应该由所有符合模型的构件所实现的那些接口。如果构件实现符合标准，则它们的运行独立于编程语言。用不同语言编写的构件可集成在同一个系统中使用。
3. 中间件为构件集成提供软件支持。为了使独立的、分布的构件一起工作，需要有处理构件之间通信的中间件。支持构件的中间件可以有效地处理低层的问题，并允许我们集中精力来处理与应用相关的问题。此外，支持构件的中间件可以提供对资源分配、事务管理、信息安全及并发的支持。
4. 开发过程适合基于构件的软件工程。你需要一个开发过程，它允许根据可用的构件功

能对需求进化演化。



CBSE 的问题

CBSE 现在是主流的软件工程方法，在创建新系统时被广泛使用。然而，当其作为一种复用方法使用特别是与其他构件集成时，会出现一些问题，包括构件可信性、构件认证、需求权衡、构件属性预测。

<http://software-engineering-book.com/web/cbse-problems/>

基于构件的开发体现了良好的软件工程实践。它不仅对于使用构件来设计系统是有意义的，即使你不是复用这些构件而是开发这些构件，也是很有意义的。基本的 CBSE 是支持构造易理解和可维护软件的有效设计原则：

1. 构件是独立的，因此它们不会影响彼此的操作。构件实现的细节是隐藏的，构件实现的改变可以不影响系统其他部分。

2. 构件通过良好定义的接口进行交互，如果这些接口能得到保持的话，构件便可以更换为另一个有更多功能或更先进功能的构件。

3. 构件基础设施提供一系列可用在应用系统中的标准服务。这减少了要开发的新代码的量。

CBSE 最初的动机是要支持复用和分布式软件工程。构件被看作是一个软件系统的元素，它可以由在不同的计算机上运行的其他构件使用远程过程调用机制来访问。每个复用构件的系统必须包含该构件的副本。这种构件的理念扩展了分布式对象的概念，就像分布式系统模型中的定义一样，如 CORBA 的规格说明（Pope 1997）。已经开发了几种不同的协议和标准支持这一对于构件的观点，例如 Sun 公司的 EJB、微软的 COM 和 .NET、CORBA 的 CCM（Lau and Wang 2007）。

不幸的是，参与提出标准的公司不能就构件的单一标准达成一致，从而限制了这种方法对软件复用的影响。对于开发的构件来说，使用不同的方法来共同工作是不可能的。针对不同的平台开发的构件，例如 .NET 或 J2EE，不能互操作。而且，提议的这些标准和协议非常复杂，很难理解。这也是采用标准和协议的一个障碍。

针对这些问题，发展了“构件即服务”的概念，并且提出标准来支持面向服务的软件工程。构件即服务的概念和构件的原始概念之间最大的不同之处是，服务是独立的实体，在使用它们的程序之外存在。当创建一个面向服务的系统时，开发者引用外部服务，而不是在系统中引入一个该服务的拷贝。

面向服务的软件工程是一种基于构件类型的软件工程。相比原先在 CBSE 中的建议，它使用了更简单的构件概念。每个使用构件的系统都嵌入了自己版本的构件。面向服务的方法正在逐渐将具有内嵌构件的 CBSE 替代为系统开发的一种方法。在本章中，将讨论具有内嵌构件的 CBSE 的使用情况；第 18 章将介绍面向服务的软件工程。

16.1 构件和构件模型

在软件复用领域，一般观点认为构件是一个独立的软件单元，可以与其他构件构成一个软件系统。然而，不同的人对软件构件提出了不同的定义。Councill 和 Heineman（Councill

and Heineman 2001) 将构件定义为:

构件是一种遵循某个标准构件模型的软件元素, 按照组装标准, 无须修改即可独立进行部署和组装^①。

这个定义实质上是基于标准的, 遵循这些标准的软件单元即为一个构件。然而, Szyperski (Szyperski 2002) 关于构件的定义却并未提及标准, 而是将重点放在了构件的关键特征上:

构件是具有契约定义的接口和显式的上下文依赖, 可独立进行部署并用于第三方组装的软件单元^②。

这两个定义都基于一个概念, 即构件作为一个元素包含在一个系统中, 而不是把构件作为一项服务被系统引用。然而, 它们还是和服务作为一个构件的概念相兼容的。

Szyperski 同时指出构件没有从外部可观察的状态, 这意味着构件的拷贝彼此是很难区分的。然而, 一些构件模型, 例如 EJB 模型, 允许有状态的构件。因此这些与 Szyperski 的构件定义不一致。虽然无状态的构件使用起来更简单, 但是存在一些系统, 其中有状态的构件的使用更加方便, 并且减少了系统的复杂性。

上面两个定义的共同性在于, 它们都认为构件是独立的并且是系统最基本的组成单元。将这些定义综合起来, 我们可以得出构件的一个更好的定义。图 16-1 展示了用于 CBSE 的构件的最重要的特性。

构件特性	描 述
可组装性	对于可组装的构件, 所有外部交互必须通过公开定义的接口进行。另外, 它还必须提供对自身信息的外部访问, 例如它的方法和属性
可部署性	为使之可部署, 构件需要是自包含的, 它必须能作为一个独立实体在提供其构件模型实现的构件平台上运行。因而意味着构件总是二进制形式的且无须在部署前编译。如果一个构件实现为一项服务, 它不必由用户来部署, 而是由服务的提供者来部署
文档化	构件必须是完全文档化的, 这样所有用户能确定是否构件满足他们的需要。应该定义所有构件接口的语法甚至语义
独立性	构件应该是独立的, 它应该可以在无其他特殊构件的情况下进行组装和部署。如果在某些情况下构件需要外部提供的服务, 应该在“请求”接口描述中显式地声明
标准化	构件标准化意味着在 CBSE 过程中使用的构件必须符合某种标准化的构件模型。此模型会定义构件接口、构件元数据、文档管理、组成以及部署

图 16-1 构件特性

考察一个构件的有效方法是将其看成独立的服务供应者, 即使构件是嵌入式的, 而不是实现为服务。当系统需要某一服务时, 会调用提供相应服务的构件, 而无须知道此构件位于何处, 也无须知道该构件是用什么程序语言开发的。例如, 图书馆系统中的构件可能提供搜索功能, 允许用户搜索不同图书馆的目录。从一种图形格式变换到另一种图形格式的构件

① Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In Component-Based Software Engineering, edited by G T Heineman and W T Councill, 5-20. Boston:Addison Wesley.

② Szyperski, C. 2002. Component Software: Beyond Object-Oriented Programming, 2nd Ed. Harlow,UK: Addison Wesley.

(如从 TIFF 格式到 JPEG 格式)可提供数据转换功能。

将构件看成服务提供者,强调的是可复用构件的两个关键特性。

1. 构件是独立可执行的实体,这是由它的接口定义的。你不必知道构件的任何源代码信息就可使用它。它可以作为一种外部服务来引用,也可以直接包含在一个程序中。

2. 构件所提供的服务可以通过其接口得到,而且所有的交互都是通过接口实现的。构件接口表示为参数化的操作,其内部状态是不会暴露出来的。

构件有两种关联接口,如图 16-2 所示。这些接口反映了构件提供的服务以及构件正确运行所需要的服务。

1. “提供”接口,定义了构件所提供的服务。这个接口是构件的 API,它定义了构件用户可以调用的方法。在一个

UML 构件图中,构件“提供”接口用一个圆圈表示,圆圈在始于构件图标的一条线段的尾端。

2. “请求”接口,指定了一个构件要进行正确的操作时系统其他构件必须提供的服务。如果这些服务不能实现,则构件将无法工作。这并不影响构件的独立性或可部署性,因为“请求”接口没有定义如何提供这些服务。在 UML 中,一个“请求”接口的标志用一个半圆形来表示,半圆位于始于构件图标的一条线段的尾端。注意:“提供”接口和“请求”接口图标相配,如同球和球洞。

为了说明这些接口,图 16-3 展示了一个构件模型,这个构件用于采集和比较来自传感器阵列的信息。它经过一段时间就自动采集数据,然后根据请求为调用构件提供比较后的数据。

“提供”接口包括对传感器进行添加、移动、开始、停止和测试等方法,report 方法返回所采集的传感器数据,listAll 方法提供所有连接的传感器的信息。尽管在这里没有给出,但这些方法自然地有相关联的参数,如定义传感器标识、位置等。

“请求”接口是用来将构件连接到传感器上的,它假设传感器有一个数据接口(通过 sensorData 访问)和一个管理接

口(通过 sensorManagement 访问)。这个接口的设计是用来连接不同类型的传感器,因而它就没有包含像 Test 和 provideReading 等这样特殊的传感器操作。相反,一个特定类型的传感器操作使用的命令嵌入在一个字符串中,这个字符串是“请求”接口中操作的参数。适配器构件解析这个字符串并翻译成嵌入的命令到每个类型传感器的具体控制接口中。本章后面将介绍适配器的用法,以及采集器构件是怎样连接到适配器上的(见图 16-12)。

使用远程过程调用(Remote Procedure Call, RPC)访问构件。每个构件都有唯一的标识符,使用这个标识符名字可以从另一台计算机上调用这个构件。被调用的构件使用相同的机制访问在其接口中定义的“请求”构件。

作为一项外部服务的构件和作为一个程序元素的构件主要的不同之处是,服务是完全独立的实体。它们没有“请求”接口。当然,它们确实需要其他构件来支持它们的操作,但是这些构件是由内部提供的。不同的程序可以使用这些服务,而无需实现任何服务所需的额外支持。



图 16-2 构件接口

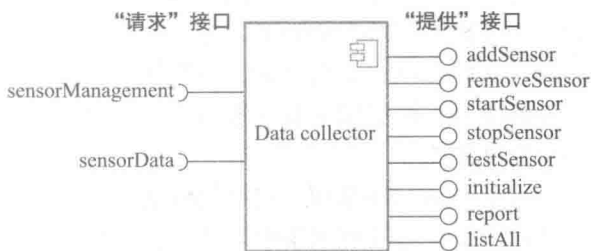


图 16-3 数据采集器构件模型



构件和对象

构件经常用面向对象语言来实现, 有时候一个构件的“提供”接口的访问是通过方法调用实现的。然而, 构件和对象类不是一个东西。与对象类不同, 构件可独立部署, 不定义类型, 与语言无关, 并且基于一个标准的构件模型。

<http://software-engineering-book.com/web/components-and-objects>

16.1.1 构件模型

构件模型定义了构件实现、文档化及开发的标准。这些标准是为构件开发者确保构件的互操作性而设立的。它们也是为那些提供中间件的构件执行基础设施供应商支持构件操作而设立的。目前已经提出了许多构件模型, 但是最重要的构件模型是现在的 Web Services 模型, Sun 公司的 EJB 模型和微软的 .NET 模型 (Lau and Wang 2007)。

Weinreich 和 Sametinger (Weinreich and Sametinger 2001) 讨论了一个理想构件模型的基本要素。图 16-4 总结了这些模型要素。该图说明, 构件模型的要素定义了构件接口、人们在程序中使用构件需知道的信息, 以及构件应该如何部署。

1. 接口。构件是通过它们的接口来定义的。构件模型规定应如何定义构件接口及在接口定义中应该包括的要素, 如操作名、参数及异常等。模型同时需指定用于定义构件接口的语言。

对于 Web 服务来说, 接口规格说明使用基于 XML 的语言, 如第 18 章所述。EJB 是 Java 专有的, 所以 Java 可用于作为接口定义语言 (Interface Definition Language, IDL); 在 .NET 中, 接口使用微软的通用中间语言 (Common Intermediate Language, CIL) 来定义。一些构件模型要求必须由构件定义专门的接口。这些接口与提供标准化服务 (如安全性和事务管理等) 的构件模型基础设施一起构成构件。

2. 使用信息。为使构件远程分布和访问, 需要给构件一个特定的名字或句柄。这个必须是全局唯一的。例如, 在 EJB 中, 有一个层次化的名字, 其根是基于因特网域名的。服务有一个唯一的统一资源标识符 (URL)。

构件元数据是构件本身相关的数据, 如构件的接口和属性信息。元数据非常重要, 用户可通过元数据发现构件提供的和所要的服务。构件模型的实现通常包括访问构件的元数据的特定方法 (如 Java 中所使用的反射接口)。

构件是通用实体, 在部署的时候, 必须对构件进行配置来适应一个应用系统。例如, 图 16-3 所示的数据采集器构件需要根据传感器阵列所限定的最多数量进行定制。因此, 构件模型应该指定如何配置二进制构件, 使其适应特定的部署环境。

3. 部署。构件模型包括一个规格说明, 此规格说明指出应该如何打包构件使其部署成为

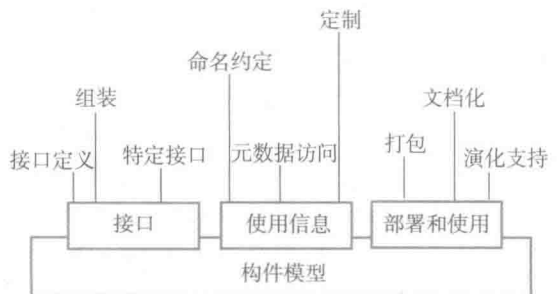


图 16-4 构件模型的基本要素

一个独立的可执行实体。由于构件是独立的实体，所以它们必须与所有支持的软件打包在一起，包括构件基础设施所不提供的以及“请求”接口未定义的软件。部署信息包括有关包中内容的信息和它的二进制构成的信息。

不可避免的是，当出现新的需求，构件就必须做出改变或者被替代。因此，构件模型应包括允许何时和怎样替换构件的控制规则。最后，构件模型定义应该产生的构件文档。这可以用于查找构件和决定构件是否适当。

对于实现为程序单元而不是外部服务的构件来说，构件模型规定了必须由支持构件执行的中间件所提供的服务。Weinreich 和 Sametinger 利用操作系统来类比解释构件模型。操作系统提供一组可被应用使用的通用服务。构件模型实现提供类似的共享服务给构件。图 16-5 给出了由构件模型实现提供的某些服务。

- 构件模型实现提供的服务包括以下两种。
1. 平台服务，允许构件在分布式环境下通信和互操作。在所有的基于构件的系统中，这些是必须有的基本服务。
 2. 支持服务，这些是很多构件都需要的共性服务。例如，许多构件需要身份认证，以确保构件服务的用户是已授权用户。提供一组标准的中间件服务供所有构件使用是有意义的。这些服务的可用性降低了构件开发的成本，而且意味着可以避免潜在的构件间不兼容。

中间件实现共性的构件服务，并提供这些服务的接口。为了利用构件模型基础设施所提供的服务，可以认为构件被部署在一个“容器”中。容器是支持服务的一个实现加上一个接口定义——构件必须提供该接口定义以便和容器整合在一起。在概念上，当你向容器添加构件时，构件可以访问支持服务，容器可以访问构件接口。将构件包含在容器中意味着构件可以访问支持服务，并且容器可以访问构件接口。在使用时，构件接口本身不能被其他构件直接访问，它们通过一个容器接口进行访问。容器接口调用代码访问内嵌构件的接口。

容器大而复杂，当在容器中部署一个构件时，能获取所有中间件服务。然而，简单的构件可能并不需要所有的中间件提供的设施。因此在 Web 服务中对共性服务供应所采取的方式相当不同。对于 Web 服务，已经为诸如事务管理和信息安全性的共性服务定义了标准，这些标准已被实现为程序库。如果你正在实现服务构件，则只能使用所需的共性服务。

与构件模型相关的服务与面向对象框架提供的设施有很多共同之处，这在第 15 章中讨论过。尽管所提供的服务可能不全面，但框架服务通常比基于容器的服务更有效率。因此，有些人认为最好使用诸如 SPRING（Wheeler and White 2013）等框架来进行 Java 开发，而不使用 EJB 中功能完整的构件模型。

16.2 CBSE 过程

CBSE 过程是软件过程，支持基于构件的软件工程。它们考虑了复用的可能性，以及在开发和使用可复用的构件中所涉及的不同过程活动。图 16-6（Kotonya 2003）给出了一个 CBSE 中过程的概览。从最高层次来说，存在两种类型的 CBSE 过程。

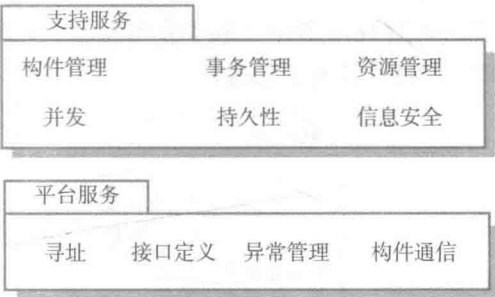


图 16-5 构件模型中所定义的中间件服务

1. 面向复用的开发。这个过程是开发将被复用在其他应用程序中的构件或服务。它通常是对已存在的构件进行通用化处理。

2. 基于复用的开发。这个过程是复用已存在的构件和服务来开发新的应用程序。

这些过程有不同的目标，因此包括不同的活动。在面向复用的开发过程中，目标是产生一个或多个可复用的构件。你必须了解将使用的构件，并且必须访问它们的源代码来将它们通用化。在基于复用的开发过程中，你不知道什么样的构件是可用的，因此你需要去发现这些构件，然后最有效地利用这些构件来设计你的系统。你可能不必访问构件的源代码。

从图 16-6 看出，基于复用和面向复用的基本 CBSE 过程支持那些与构件获得、构件管理、构件认证有关的过程。

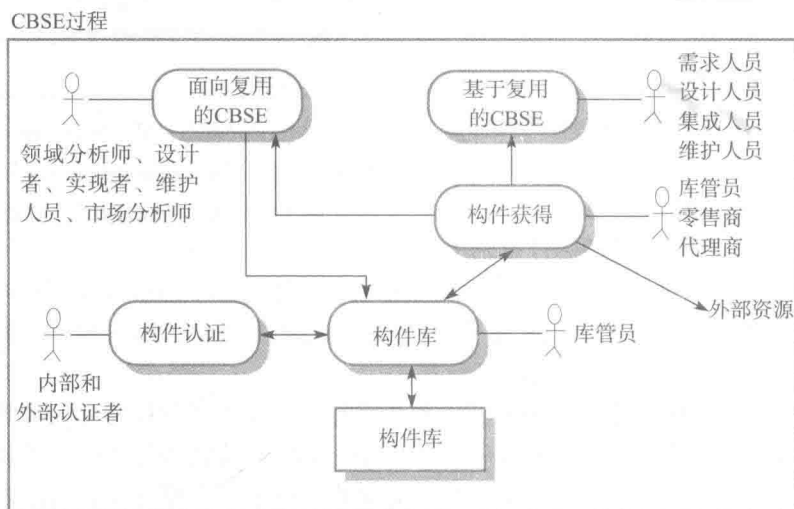


图 16-6 CBSE 过程

1. 构件获得，是得到面向复用的构件或开发一个可复用的构件的过程。它可能涉及获得内部开发的构件或服务，或从外部资源找到这些构件。

2. 构件管理，是管理一个企业的可复用的构件，以确保它们得以正确分类和存储，使面向复用成为可能。

3. 构件认证，是检查构件和证实这个构件符合其规格说明的过程。

一个组织机构所持有的构件可能会存储在构件库中，其中包括构件和它们的使用信息。

16.2.1 面向复用的 CBSE

面向复用的 CBSE 是开发可复用的构件和通过一个构件管理系统使对它们的复用成为可能的过程。早期的 CBSE 的支持者（Szyperski 2002）的观点是：发展一个繁荣的构件市场，将有专门的构件供应商和构件厂商，他们根据不同的开发商来组织构件的销售。软件开发商购买构件以包含在系统中或是为使用他们的服务而付费。然而，这个观点并没有变成现实。目前存在相当少的构件供应商，购买构件也是很少见的。

结果，面向复用的 CBSE 最有可能发生在一个提倡复用驱动的软件工程的组织机构中。这些公司有内部开发的可复用构件库。然而，这些内部开发的构件如果不改变，通常是不可

复用的。它们经常包含特定应用程序的特征和接口，这些在其他的构件复用的程序中可能是不需要的。

为了使构件可复用，要改写和拓展这些构件以创建更通用的、更适合复用的版本。显然，这有一个相关的成本问题。首先，必须决定构件是否能复用；其次，未来复用节约的成本是否能抵得上使构件可复用的成本。

为回答第一个问题，必须决定是否构件实现一个或多个稳定的领域抽象。稳定的领域抽象是应用领域中变化缓慢的基本概念。例如在银行系统中，领域抽象包括账户、账户持有者和账目。在医院管理系统中，领域抽象可能包括病人、治疗和护士。这些领域抽象有时称为业务对象。如果构件是对普遍使用的领域抽象或是一组相关对象的实现，它大概就可能被复用。

为回答有关成本的问题，必须评估为使构件可复用而需要进行的变更的成本。这些成本包括构件文档化的成本、构件可靠性验证的成本以及使构件更通用的修改成本。提高构件可复用性的变更包括：

- 去除那些应用特定的方法；
- 更名使其更通用；
- 添加方法提供更完备的功能覆盖；
- 为所有方法构造一致的异常处理过程；
- 添加“配置”接口，允许对构件进行调整以适应不同的使用情况；
- 集成必要的构件以增强独立性。

异常处理是一个十分困难的问题。原则上，所有异常应该作为构件接口的一部分。构件不应该处理自身的异常，因为每个应用程序都有自己对异常处理的需求。更确切地说，构件应定义会产生的异常并将之发布为接口的一部分。例如，有一个实现堆栈数据结构的简单构件，应检测和发布栈上溢和下溢的异常。然而实际情况是，在这个过程中存在以下两个问题。

1. 发布所有的异常将导致接口膨胀，这将更加难以理解。这可能会丢掉构件的潜在用户。

2. 构件的运行可能依靠局部异常处理，改变它将严重影响构件的功能。

因此，对于构件的异常处理，必须采取一个实用的方式。常见的技术异常应当被局部处理，恢复构件的运行很重要。这些异常以及如何对异常进行处理应该文档化。其他有关构件业务功能的异常应当传递给调用组件进行处理。

Mili 等人 (Mili et al. 2002) 讨论了对开发可复用构件的成本及投资所能带来的收益进行估计的方法。复用构件较之重新开发构件不仅仅是生产率问题，还包括质量收益。因为可复用构件更可靠，且具有市场时效性。这些额外的收益来自于更快速地部署软件。

Mili 等人提出各种公式来估算这些收益，正如第 23 章讨论的 COCOMO 模型那样。然而，这些公式的参数很难准确估计，并且公式必须根据具体环境进行调整，使得使用它们很困难。可能只有极少软件项目管理者使用这些模型去估计从构件可复用性的投资中得到的回报。

一个构件是否可复用依赖于它的应用领域、功能性和通用性。如果其应用领域常见，并且这个构件实现了这个领域的标准功能，那么这个构件更可能具有可复用性。当我们使构件具有更多的通用性的时候，其可复用性也在提高。然而，这也使得构件具有更多的操作，更

为复杂，更难以理解和使用。

构件的可复用性与构件的可理解性之间不可避免地需要折中。为了使构件可复用，必须提供一组通用接口和操作，以满足构件的所有可能的使用方式。可复用性增加了复杂性，同时也降低了构件的可理解性，因而决定何时和怎样复用构件是很困难的事情。由于了解可复用构件需要时间，有时候重新实现一个具有必要特定功能的更简单的构件更具成本效益。

构件的另一潜在的来源是现存的遗留系统。如第 9 章所讨论的，遗留系统完成了很重要的业务功能，但却是用过时的软件技术编写的。其结果是很难将之与新系统一起使用。但如果将这些旧系统转化成构件，它们的功能就可以在新的应用中使用了。

当然，这些遗留系统通常没有清晰定义的“请求”和“提供”接口。为使这些构件可复用，必须对它进行封装，由此定义构件的接口。该封装将原代码的复杂性隐藏了起来，并为外部构件访问的服务提供了接口。虽然此封装就是一个相当复杂的软件，因为它必须访问遗留系统的功能，然而，对封装的开发成本通常远低于对遗留系统重新实现的成本。

一旦你开发和测试一个可复用的构件或服务，必须针对未来的复用来管理这个构件或服务。管理包括：如何对构件进行分类以便可以发现它；使现有的构件是可用的，无论它是作为一个存储库还是作为一个服务；维护构件的使用信息；保存不同的构件版本的记录。如果这个构件是开源的，可以在一个公开的存储库例如 GitHub 或者是 Sourceforge 中使它成为可用的。如果是计划在业内使用，你可能要使用一个内部的存储库系统。

做复用项目的公司在构件成为可复用构件之前，要执行某种形式的构件认证。认证意味着除了开发者之外，可能有些人要检查这个构件的质量。在构件成为可复用构件之前，测试构件以确保达到一个可以接受的质量标准。然而，这可能是个非常昂贵的过程，并且许多公司简单地将测试和质量检查工作留给构件的开发者。

16.2.2 基于复用的 CBSE

成功的构件复用需要一个经过裁减的合适的开发过程，以便在软件开发过程中包含可复用的构件。基于复用的 CBSE 过程必须包括搜索和集成可复用构件的活动。这种过程的结构在第 2 章讲过，图 16-7 给出了在 CBSE 过程中的主要活动。这个过程某些活动，如用户需求的最初发现，其执行方式与在其他软件过程中的执行方式是相同的。然而，基于复用的 CBSE 与先前的软件开发过程之间存在以下主要不同点。

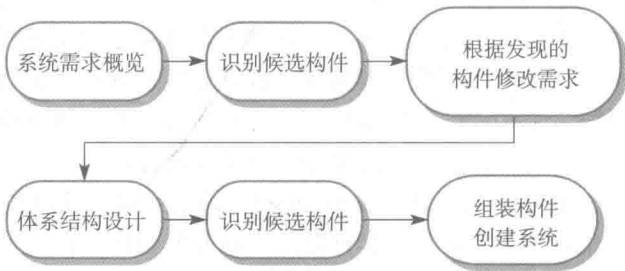


图 16-7 基于复用的 CBSE

1. 最初开发的用户需求只需是概要性的而不是十分详细的，且鼓励利益相关者在定义他们的需求时尽可能灵活。太特殊的需求限制了能满足这种需求的构件数量。然而，不像增量式开发，我们需要完整的需求，这样就能尽可能多地识别出可复用的构件。
2. 在过程的早期阶段根据可利用的构件来细化和修改需求。如果可利用的构件不能满足用户需求，就应考虑可以由复用构件支持的相关需求。如果这意味着能节省开支且能快速地开发系统，或许用户愿意改变想法。

3. 在系统体系结构设计完成后, 会有一个进一步的对构件搜索及设计精化的活动。一些似乎可用的构件会变得不合适或不能与其他已选构件一起正常工作。你可能不得不针对这些构件找到一些替代方案。这些构件的功能可能还需要适当进行修改。

4. 开发就是将已发现的构件集成在一起的组装过程。其中包括将构件与构件模型基础设施集成在一起, 有时, 包括开发适配器来协调不匹配的构件的接口。当然, 除了复用构件所提供的功能外还可能添加其他的功能。

体系结构设计阶段特别重要, Jacobsen 等 (Jacobsen, Griss, and Jonsson 1997) 发现定义一个鲁棒的体系结构对于成功的复用具有决定性的意义。在这个体系结构设计阶段, 将选择一个构件模型和一个实现平台。然而, 许多公司有一个标准的开发平台 (例如 .NET), 因此这个构件模型是提前决定的。如第 6 章所讲过的, 要建立系统的一个高层体系结构, 并对系统的分布和控制做出决断。

CBSE 过程的一个独特活动是为复用来识别候选构件或服务。这包括一系列子活动, 如图 16-8 所示。首先, 你的注意力集中于搜索和选择, 确信有可复用构件能满足需求。显然, 应做些初始的检查看构件是否合适, 但不需要做详细测试。在后期阶段, 系统体系结构设计完成后, 应将更多的时间用于构件确认上。你需要确信所识别出的构件真正适合你的应用, 若不能, 就必须重复搜索和选择过程。

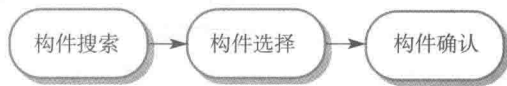


图 16-8 构件识别过程

识别构件的第一个阶段是从你的公司或者可靠的供销商那里得到可用的构件。正如前面提到的, 构件供应商相对较少, 因此你很可能需要要在自己的公司寻找已开发的构件或者开源软件库。软件开发公司可以建立自己的可复用构件库, 而不必冒险使用外部供应商的构件。或者, 你可能决定在 Web 上搜索源码库, 例如 Sourceforge、GitHub、Google Code, 看看你需要的这个构件的源码是否是可用的。

一旦构件搜索过程找到可能的构件, 你必须挑选出候选构件。有些情况下, 这是个比较容易完成的任务——列表上的构件直接实现用户需求, 而没有竞争构件能匹配这些需求。然而在另外一些情况下, 选择过程是非常复杂的。构件和需求之间没有一个清晰的映射关系。你可能发现必须用多个构件集成在一起才能满足一个特定的需求或一组需求。所以你必须决定哪种构件组装能最好地覆盖你的需求。

选取好系统可能包含的构件后, 应对它们进行确认以检查它们是否像广告宣传的一样。确认的程度有赖于构件来源。如果所用构件来自熟悉的可靠厂商, 就没必要逐个测试构件性能, 只需对构件做集成测试即可。相反, 如果使用一个来自不熟悉的供应商的构件, 在其加入系统之前必须坚持对其进行检查和测试。

构件确认包括对构件开发一组测试用例 (或者可能的话, 拓展随构件提供的测试用例), 并开发测试程序去运行构件测试。构件测试的主要问题是, 构件的规格说明可能不够详细, 不足以允许你开发一个全面的构件测试集。构件的规格说明总是非正规的, 唯一标准的可能就是其接口规格说明。这样构件的信息就不足以让你开发完全的测试集, 因此你很难确信声明的构件接口正是你所需要的。

在测试可复用构件是你所需要的同时, 你还必须确定这个构件不包括任何恶意的代码或你不需要的功能。专业的开发者很少使用不可靠来源的构件, 尤其是当这些资源没有提供源代码时。因此, 恶意代码问题通常不会发生。然而, 构件可能通常包括你不需要的功能, 你

必须检查这个功能是否会阻碍这个构件的使用。

不需要的功能这个问题可能是构件本身产生的。这个可能使构件变得缓慢，导致它产生出乎意料的结果，或者在某种情况下，导致严重的系统失效。图 16-9 总结了一个在复用的系统中那些不需要的功能导致灾难性软件失效的情况。

阿丽亚娜 5 型运载火箭失效

在开发阿丽亚娜 5 型空间运载火箭的时候，设计者决定复用在阿丽亚娜 4 型中非常成功的惯性参照系软件。此惯性参照系软件维持火箭的稳定。他们决定不加改变地复用此软件（一般人们都会这么做），尽管它包含了额外的功能，超出了阿丽亚娜 5 型的需要。

在阿丽亚娜 5 型首次发射中，惯性导航软件在升空后失败，火箭因而失去控制。地面控制人员向火箭发送指令令其自毁，于是火箭及其有效载荷被摧毁。事后的调查发现，导致问题的原因是某个定点数到整数转换中发生了数的溢出，这是个未处理的异常。这导致运行系统关闭了惯性参照系软件系统，所以火箭的稳定性得不到维持。此故障在阿丽亚娜 4 型中没有发生是因为它的引擎功率比较小，所转换的值不会大到溢出的程度。

这表明了软件复用的一个重要问题。软件是基于系统将会被使用的上下文环境的假设而开发的，而这些假设在一个不同的复用情形下可能与实际不符。

关于此失效的更多信息可以在 <http://software-engineering-book.com/case-studies/ariane5/> 找到。

图 16-9 复用软件确认失效的例子

阿丽亚娜 5 型运载火箭出现问题是因为对阿丽亚娜 4 型的软件所做的假设对于阿丽亚娜 5 型是不适用的。这是可复用构件的常见问题。它们最初是针对某个应用环境实现的，自然地就嵌入了对那个环境的假设。这种假设很少被文档化，因此，当构件被复用，不可能设计测试来检查是否假设仍然是有效的。如果你在新的环境中复用一个构件，你可能不会发现内含的环境假设，直到你在一个运行系统中使用这个构件。

16.3 构件组装

构件组装是指构件相互直接集成或是用专门写的“胶水代码”将它们整合在一起创造

一个系统或另一个构件的过程。组装构件有很多种不同的方法，正如图 16-10 中描述的那样。从左到右，图中展示了顺序组装、层次组装和叠加组装。在下面的讨论中，假设使用两个构件（A 和 B）来组装一个新的构件。

1. 顺序组装。通过按顺序调用已经存在的构件，你可以用两个已经存在的构件来创造一个新的构件。你可以把这个组装看作是“提供”接口的组装。也就是说，调用构件 A 提供的服务，然后用 A 返回的结果调用构件 B 提供的服务。在顺序组装中，构件间不会相互调用，但是可能会被外部程序调用。这种组装的类型可能适用于作为程序元素的构件或是作为服务的构件。

需要特定的胶水代码来按照正确的顺序调用构件服务以及确保构件 A 提供的结果和构件 B 所期望的输入相兼容。胶水代码将这些输出转化成构件 B 需要的输入。

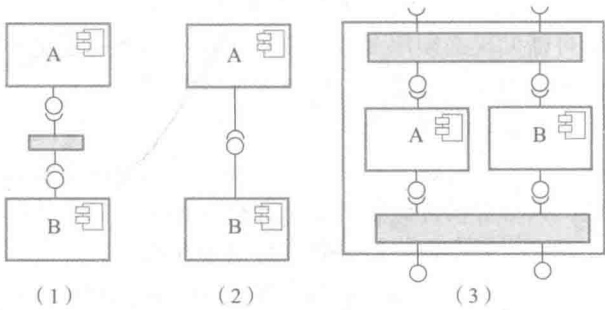


图 16-10 构件组装的类型

2. 层次组装。这种情况发生在一个构件直接调用由另一个构件所提供的服务时。被调用的构件为调用的构件提供所需要的服务。因此，被调用构件的“提供”接口必须和调用构件的“请求”接口兼容。

构件 A 直接调用构件 B，如果它们的接口相匹配，则构件 A 可以直接调用构件 B，这就不需要额外的代码。然而，如果构件 A 的“请求”接口和构件 B 的“提供”接口不匹配，则需要一些转换代码。因为服务没有“请求”接口，当构件作为网络服务来实现时，是不能使用层次组装模式的。

3. 叠加组装。这种情况发生在两个或两个以上构件放在一起（叠加）来创建一个新构件的时候，这个新构件合并了它们的功能。这个新构件的“提供”接口和“请求”接口是构件 A 和构件 B 各自对应的接口的一个组合。通过组装构件的外部接口来分别调用构件 A 和 B。A 和 B 不相互依赖，也不相互调用。这种组装类型适合于构件是程序单元或者构件是服务的情形。

当创建一个系统时，可能用到所有形式的构件组装方式。对所有情况，你都必须写“胶水代码”来连接构件。例如在顺序组装中，构件 A 的输出成为构件 B 的输入，这就需要一个中间声明来调用构件 A，收集结果然后用该结果作为参数来调用构件 B。当一个构件调用另外一个构件时，你可能需要引入一个过渡构件以确保“提供”接口和“请求”接口是兼容的。

当你编写构件尤其是为了组装来写构件时，必须注意设计好构件接口以使其在这个系统中相互匹配。这样我们就可以很容易地把这些构件组装为一个单元。然而，当独立开发可复用构件时，你经常会面临接口不兼容的问题，即所要组装的构件的接口不一致的问题。会发生 3 种不兼容情况。

1. 参数不兼容。接口每一侧的操作有相同的名字，但参数类型或参数数目是不同的。在图 16-11 中，addressFinder 返回的 location 参数和 mapDB 中的 displayMap 和 printMap 要求的参数并不相互兼容。

2. 操作不兼容。“提供”接口和“请求”接口的操作名不同。这也是图 16-11 所示构件之间的更进一步的不兼容。

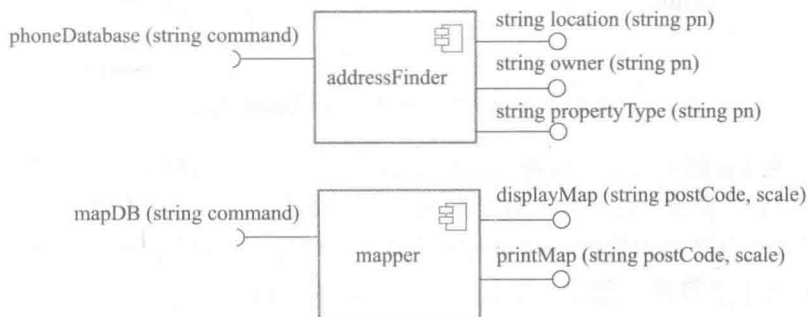


图 16-11 带有不兼容接口的构件

3. 操作不完备。一个构件的“提供”接口是另一个构件的“请求”接口的一个子集，或者相反。

针对所有情况，都必须通过编写适配器构件来解决不兼容的问题，适配器构件使两个可复用构件的接口相一致。适配器构件将一个接口转换为另外一个接口。

适配器的准确形式依赖于组装的类型。例如，在下面的例子中，适配器仅是从某个构件得到结果，然后将其转化为另一构件的输入。在其他情况，适配器可能作为构件 B 的代理被构件 A 调用。这种情况发生在如果 A 希望去调用 B，但是 A 的“请求”接口细节和 B 的“提供”接口细节不匹配的情形。适配器通过将来自 A 的输入参数转换为 B 需要的输入参数来协调这些差异。然后就可调用 B，将服务传达给 A 了。

为了说明适配器，考虑图 16-11 中的构件，它们的接口是不兼容的。这可能是应急服务中所使用的系统的一部分。当应急操作员接听呼叫时，电话号码被输入到 addressFinder 构件来定位地址。然后，用 mapper 构件打印出地图发给分派到应急现场的车辆。事实上，构件的接口比这里给出的要复杂得多，但是这个简化的版本有助于说清楚适配器的概念。

第一个构件 addressFinder 查找与电话号码相匹配的地址。它还能返回该电话号码相应的房产主人。mapper 构件使用邮政编码（在美国为标准 ZIP 码附加 4 位标识房产位置的阿拉伯数字），以某个比例尺显示或打印编码所代表的相关地区的街区地图。

原则上这些构件是兼容的，因为房产位置包括邮编或 ZIP 码。然而，你必须写一个叫作 postCodeStripper 的适配器构件，它从 addressFinder 得到位置数据并剥离出邮政编码。这一邮政编码转而作为 mapper 的输入，街区地图以 1 : 10 000 的比例显示。下面的代码说明了实现它所需要的调用序列：

```
address = addressFinder.location(phonenummer);
postCode = postCodeStripper.getPostCode(address);
mapper.displayMap(postCode, 10000);
```

使用适配器构件的另一种情况是在层次组装中，即某一构件需利用另一构件，而两者的“提供”接口和“请求”接口是不兼容的。图 16-12 说明了适配器的用法，这个适配器是用来连接数据采集器和传感器的。它们会用在野外气象站系统的实现中，正如第 7 章讨论的。

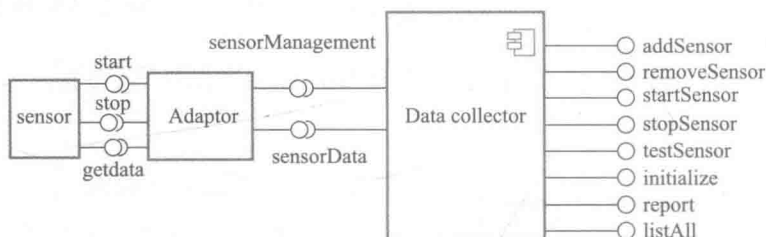


图 16-12 连接数据采集器和传感器的适配器

传感器构件和数据采集器构件使用适配器组装。适配器使数据采集器构件的“请求”接口与传感器构件的“提供”接口相一致。数据采集器构件在设计上是采用一种通用的“请求”接口，这个接口支持传感器数据采集和传感器管理。对每一个操作来说，它的参数代表具体传感器指令的文本字符串。例如，发布一个采集命令，你必须写 sensorData("collect")。正如图 16-12 中所表示的，传感器本身有单独的操作，如 start、stop 和 getdata。

适配器解析输入的串，识别命令（例如，collect），然后调用 Sensor.getdata 来得到传感器的值。然后它将结果（作为字符串）返回给数据采集器构件。这个接口的风格意味着数据采集器可以和不同类型的传感器交互。对于每个类型的传感器要实现一个单独的适配器，它可以将传感器指令从 Data collector 转换到实际的传感器接口。

前面关于构件组装的讨论假设你可以从构件文档中判断出接口是否兼容。当然，接口的

定义包括操作名和参数类型，因此可根据这些评估兼容性。然而，却不能够依赖构件文档判断出接口是否语义兼容。

为了说明这个问题，考虑图 16-13 中给出的组装例子。这些构件是用来实现一个能从数码相机传输图像文件并将它存储在图片库中的系统。系统用户可以提供额外的信息来描述图片和对图片制作目录。为避免混乱，这里并没给出所有的接口方法，只简单给出几个方法，因为需要它们来解释构件文档化问题。Photo Library 接口方法是：

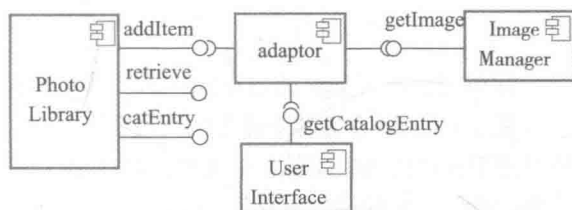


图 16-13 图片库组装

```

public void addItem(Identifier pid; Photograph p; CatalogEntry photodesc);
public Photograph retrieve(Identifier pid);
public CatalogEntry catEntry(Identifier pid);
  
```

假设 Photo Library 中的 addItem 方法的文档如下：

此方法将图片加入图片库，并将图片标识符和目录描述符与图片关联起来。

此描述看起来是解释构件做什么，但是让我们考虑如下问题：

- 若图片标识符已与库中图片关联会怎样？
- 图片描述符与目录项和图片是关联的吗？也就是说，如果删除图片，是否也要删除目录信息？

在 addItem 的非正式描述中是没有足够的信息回答这些问题的。当然，可以添加更多的信息到方法的自然语言描述中，但是，总的来说，解决这种二义性的最好方法是使用形式化语言来描述接口。图 16-14 所给出的描述是 Photo Library 的接口描述的一部分，它是对非形式化描述添加了一些信息。

```

- The context keyword names the component to which the conditions apply
context addItem

- The preconditions specify what must be true before execution of addItem
pre:   PhotoLibrary.libSize() > 0
       PhotoLibrary.retrieve(pid) = null

- The postconditions specify what is true after execution
post:  libSize() = libSize()@pre + 1
       PhotoLibrary.retrieve(pid) = p
       PhotoLibrary.catEntry(pid) = photodesc

context delete

pre:   PhotoLibrary.retrieve(pid) <> null ;

post:  PhotoLibrary.retrieve(pid) = null
       PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
       PhotoLibrary.libSize() = libSize()@pre-1
  
```

图 16-14 Photo Library 接口的 OCL 描述

图 16-14 中的描述使用了前置条件和后置条件，这是基于对象约束语言（Object Constraint Language, OCL）定义的一套符号，而 OCL 是 UML 的一部分（Warmer and Kleppe 2003）。OCL 是设计用来描述 UML 对象模型中的约束的，它允许你表达必为真的谓词，或者是在执

行某个方法之前必为真的谓词，或者是在执行完某个方法后必为真的谓词。这些都是不变量、前置条件和后置条件。为了在操作前访问一个变量的值，可以在它的名字后加上 @pre。下面使用 age 作为一个例子：

```
age = age@pre + 1
```

此语句意味着 age 的值在操作后要比操作前多 1。

基于 OCL 的方法越来越多地使用在为 UML 模型加入语义信息，OCL 描述在模型驱动工程中可用于驱动代码生成器。一般方法源自 Meyer 的“契约设计”方法 (Meyer 1992)，在这种方法中，通信对象的接口和责任（或者说契约）是形式化定义的，并且被运行时系统执行。Meyer 认为，如果我们要开发可靠的构件，那么“契约设计”方法就是绝对必要的 (Meyer 2003)。

图 16-14 包括了 Photo Library 中 addItem 方法和 delete 方法的规格说明。所定义的方法是通过关键词 context 来指示的，前置条件和后置条件是用 pre 和 post 来指示的。

addItem 的前置条件陈述为：

1. 图片库中的图片不能具有与待加入图片相同的标识符。
2. 库必须是存在的——假定创建一个库就加入一个项在其中，这样库的大小始终大于 0。

addItem 的后置条件陈述为：

1. 库的大小增 1 (因此只产生一项)。
2. 如果用同一个标识符检索图片，就可以得到新添加的那个图片。
3. 如果再次用相同的标识符在目录中查找，就可以得到所生成的目录项。

delete 的规格说明提供更多的信息。前置条件陈述为：要删除一个项，它必须在库中，删除图片后不能再检索到它，且库的大小要减 1。然而，delete 并不删掉目录项——图片被删除后仍然可以检索到它。其原因在于，你可能希望在目录中维持关于图片为何被删除、它的新位置等信息。

当通过组装构件来创建一个系统时，你会发现在功能性需求和非功能性需求之间，在尽可能快速移交软件和创建一个能随需求变更而进化的系统之间，都存在着潜在的冲突。你将必须做出折中的地方包括：

1. 在交付系统功能性需求方面哪种构件组装方式是最有效的？
2. 哪种构件组装方式可以使组装的构件在需求变更时更容易做出调整？
3. 组装系统将有哪些总体特性？这些总体特性包括性能和可依赖性。只有当整个系统实现后你才能够评估这些特性。

不幸的是，解决组装问题的方案在许多情况下可能是相互冲突的。例如，考虑如图 16-15 中所描述的情况，系统可通过两种可选组装方式而创建。该系统是一个数据采集和报告系统，数据从不同的数据源采集而来，存储到一个数据库，然后可以产生不同汇总数据的报告。

这里，在适应性和性能两者之间是有潜在冲突的。图 16-15a 所示组装方式适应性更强，而图 16-15b 所示组装方式也许更快和更可信。图 16-15a 所示组装方式的优

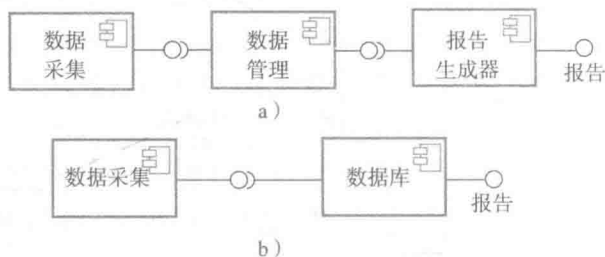


图 16-15 数据采集和报告生成构件

势在于报告和数据管理是分离的，因此应对未来的变更更为灵活。数据管理系统可以被替换，且如果需要报告而现有的报告系统不能生成时，构件同样可被替换而不必改变数据管理构件。

图 16-15b 所示组装方式中，使用的是内嵌报告设施（如微软的 Access）的数据库构件，该组装方式的主要优点在于构件很少，因此就可能很快实现，因为没有构件间太多的通信负载。此外，应用于数据库的数据完整性规则同样也应用于报告。这些报告不可能以不正确的方式组装数据。在图 16-15a 所示组装方式中没有这种限制，因此在报告中就更容易出错。

总的说来，一种好的组装原则是“关注点分离”。也就是说，应该用这样的方式设计你的系统，即每个构件都有清晰定义的任务，理想情况是，这些任务彼此不会重叠。然而，购买一个多功能的构件会比购买两个或三个独立构件更便宜。而且，使用多个构件在可依赖性或在性能上可能会有损失。

要点

- 基于构件的软件工程（CBSE）是一种基于复用的方法，用这种方法来定义、实现和组装松散耦合的独立构件，使之成为一个系统。
- 构件是一个软件单元，它的功能和可依赖性完全由一套公共接口定义。构件可以与其他构件相组装而无须考虑它们的实现细节，而且可以部署为一个可执行单元。
- 构件可能实现为程序单元进而被包含在系统当中，或是实现为在系统中引用的外部服务。
- 构件模型定义了一组构件标准，包括接口标准、使用标准和部署标准。构件模型的实现提供了一套公共服务，可以为所有构件所用。
- 在 CBSE 过程中，我们不得不将需求工程过程与系统设计交织在一起，不得不在所希望的需求和从现存的可复用构件能得到的服务之间采取折中。
- 构件组装是将构件“捆”到一起来创建一个系统的过程。组装的类型包括顺序组装、层次组装及叠加组装。
- 当组装不是专为你的应用写的可复用构件时，通常要编写适配器或“胶水代码”，以便使不同构件接口互相兼容。
- 当选择组装方式时，必须考虑系统所需的功能性需求、非功能性需求，以及当系统发生改变时，一个构件能被另一构件代替的难易。

阅读推荐

《Component Software: Beyond Object Oriented Programming, 2nd ed》讨论了 CBSE 中的技术和非技术问题。与 Heineman 和 Councill 的书相比，本书更详细地讨论了专业的技术，还包括对市场问题的透彻讨论。（C Szyperski, Addison Wesley, 2002）

《Specification, implementation and deployment of components》是一篇非常好的关于 CBSE 的基本教程。CACM 的同一期中包括构件和基于构件开发的论文。（I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, Comm. ACM, 45 (10), October 2002）<http://dx.doi.org/10.1145/570907.570928>

《Software Component Models》对商用和研究用的构件模型进行了全面的讨论，它将这些模型进行分类，并解释了它们之间的差异。（K-K. Lau and Z. Wang, IEEE Transactions on

Software Engineering, 33 (10), October 2007) <http://dx.doi.org/10.1109/TSE.2007.70726>

《 Software Components Beyond Programming: From Routines to Services 》是一个专刊的开篇文章, 这期专刊有几篇关于软件构件的文章。这篇文章讨论了构件的演化以及面向服务的构件正在如何取代可执行编程例程。(I. Crnkovic, J. Stafford, and C. Szyperski, IEEE Software, 28 (3), May/ June 2011) <http://dx.doi.org/10.1109/MS.2011.62>

《 Object Constraint Language (OCL) Tutorial 》是对于对象约束语言使用的很好的介绍。(J. Cabot, 2012) <http://modeling-languages.com/ocl-tutorial/>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap16/>

支持视频的链接: <http://software-engineering-book.com/videos/software-reuse/>

关于阿丽亚娜 5 型运载火箭事故的详细讨论: <http://software-engineering-book.com/case-studies/ariane5/>

练习

- 16.1 为什么对所有构件的交互都通过“请求”接口和“提供”接口来定义是重要的?
- 16.2 构件独立原则表示用完全不同的方法实现的一个构件代替另一个构件是有可能的。举例说明这样的构件代换可能会出现不希望的后果, 且会导致系统失效。
- 16.3 构件作为程序元素与作为服务之间有什么根本不同?
- 16.4 为什么构件应基于标准构件模型是很重要的?
- 16.5 列举一个实现抽象数据类型(如堆栈或链表)的构件的例子, 说明为什么通常可复用构件总是需要扩展和改写。
- 16.6 为什么没有构件源代码实现可复用构件的确认是相当困难的? 以怎样的形式化构件规格说明才能简化确认问题?
- 16.7 设计一个可复用构件的“提供”接口和“请求”接口, 这个构件在第 1 章的 Mentcare 系统中可能用于表示一个病人。
- 16.8 举例说明需要不同类型的适配器来支持顺序组装、层次组装和叠加组装。
- 16.9 设计构件接口, 该构件可能用于应急控制室的系统中。为 call-logging 构件设计接口, 该构件记录所发生的呼叫; 再为 vehicle discovery 构件设计接口, 该构件在给定邮政编码 (ZIP 码) 和事故类型后, 找出最近的可用车辆并分派到事故现场。
- 16.10 有人提出应建立独立的认证机构。供应商可以提交他们的构件给该机构, 由此机构验证它的可信任性。这种认证机构的优点和缺点是什么?

参考文献

Councill, W. T., and G. T. Heineman. 2001. “Definition of a Software Component and Its Elements.” In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 5–20. Boston: Addison-Wesley.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

Kotonya, G. 2003. “The CBSE Process: Issues and Future Visions.” In *2nd CBSEnet Workshop*. Budapest, Hungary. <http://miro.sztaki.hu/projects/cbsenet/budapest/presentations/Gerald-CBSEProcess.ppt>

- Lau, K-K., and Z. Wang. 2007. "Software Component Models." *IEEE Trans. on Software Eng.* 33 (10): 709–724. doi:10.1109/TSE.2007.70726.
- Meyer, B. 1992. "Applying Design by Contract." *IEEE Computer* 25 (10): 40–51. doi:10.1109/2.161279.
- . 2003. "The Grand Challenge of Trusted Components." In *Proc. 25th Int. Conf. on Software Engineering*, Portland, OR: IEEE Press. doi:10.1109/ICSE.2003.1201252.
- Mili, H., A. Mili, S. Yacoub, and E. Addy. 2002. *Reuse-Based Software Engineering*. New York: John Wiley & Sons.
- Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.
- Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Harlow, UK: Addison-Wesley.
- Warmer, J., and A. Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston: Addison-Wesley.
- Weinreich, R., and J. Sametinger. 2001. "Component Models and Component Services: Concepts and Principles." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 33–48. Boston: Addison-Wesley.
- Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.

分布式软件工程

目标

本章的目标是介绍分布式系统工程和分布式系统体系结构。阅读完本章后，你将：

- 了解在设计和实现一个分布式软件系统时不得不考虑的关键问题；
- 理解客户 - 服务器计算模型和客户 - 服务器系统的分层体系结构；
- 了解分布式系统体系结构的常用模式，以及不同类型的系统所适合的体系结构模式；
- 理解软件即服务的概念，提供了对远程应用程序的基于 Web 的访问。

大多数基于计算机的系统现在是分布式系统。分布式系统涉及多台计算机，而不是单个机器上运行的单个应用程序。即使是看起来像在 PC 或笔记本电脑上运行的独立应用（如图片编辑器），也可能是分布式系统。它们在一个单一的计算机系统中运行，但经常依赖于远程云系统的更新、存储和其他服务。Tanenbaum 和 Van Steen（Tanenbaum and Van Steen 2007）给出了分布式系统的定义：一个独立计算机的集合体，给用户的感觉却是面对单个完整的系统^①。

当设计一个分布式系统时，由于系统是分布式的，有一些具体的问题必须考虑。考虑这些问题是因为系统的不同部分在独立管理的计算机上运行；而且，在设计中，网络的特性（例如延迟和可靠性）可能也需要考虑到。

Coulouris 等人（Coulouris et al. 2011）发现了用分布式方法开发系统的如下 5 个优势。

1. 资源共享。分布式系统允许硬件、软件资源（如磁盘、打印机、文件和编译器等）共享使用，这些资源与网络上的计算机相关联。

2. 开放性。分布式系统通常都是开放系统——系统围绕标准的互联网协议进行设计，从而使来自不同供应商的设备和软件可以结合起来。

3. 并发性。在分布式系统中，多个进程可以在网络的不同计算机上同时运行。这些进程在其正常运行期间可以（但不是必需）彼此通信。

4. 可伸缩性。原则上，分布式系统至少要有可伸缩性，系统的能力可以通过增加新的资源来满足对系统的新的需求。实际过程中，可伸缩性可能受到网络的限制。

5. 容错性。多台计算机及其信息复制能力意味着系统对硬件和软件失效具有相当的容错能力（见第 11 章）。在绝大多数分布式系统中，失效的发生可能会使提供服务的能力下降，但彻底丧失服务能力只有在发生网络失效时才会出现^②。

分布式系统天然比集中式系统更复杂。这使得分布式系统更难设计、实现和测试。理解分布式系统的涌现特性更加困难，因为系统构件以及系统基础设施之间的交互很复杂。例如，系统性能取决于网络带宽、网络负载以及系统中其他计算机的速度，而不再是单个处理

① Tanenbaum, A.S., and M. Van Steen. 2007. Distributed Systems: Principles and Paradigms, 2nd Ed. Upper Saddle River, NJ: Prentice-Hall.

② Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. Distributed Systems: Concepts and Design, 5th Edition. Harlow, UK: Addison Wesley.

器的速度。从系统的一个部分移动资源到另一个部分可能会对系统性能产生显著的影响。

此外,正如 WWW 的所有用户所了解的,分布式系统的响应是不可预知的。响应取决于系统的总负载、系统体系结构以及网络负载。由于这些因素都是在短时间内快速变化的,同一个用户的两个请求之间可能会有很大差别。

在过去的几年里,影响分布式软件系统的最重要的进展是面向服务系统和云计算、基础设施即服务、平台即服务、软件即服务的出现。本章的重点是分布式系统的一般问题,但是,17.4 节会涉及软件即服务的概念,这与第 18 章的内容是呼应的。第 18 章的重点是面向服务软件工程的其他类似问题。

17.1 分布式系统

正如本章的引言中提到的,分布式系统比在单处理器上运行的系统要复杂得多。这种复杂源于分布式系统不可能有一个自顶向下的控制模型。系统中提供功能的节点通常是独立的系统,它们没有单独管理系统的权限。连接这些节点的网络是一个单独管理的系统。这个系统本身是很复杂的,并且不能被使用网络的用户所控制。因此,分布式系统的运行本质上就会有不可预测性,系统的设计者必须要考虑到这一点。

在分布式系统工程中,有一些不得不考虑的重要的设计问题。

1. 透明性。分布式系统在多大程度上应该向用户呈现为一个单个系统?对于用户何时了解系统的分布性是有益的?

2. 开放性。系统是否应该使用支持互操作性的标准协议来设计?是否应该使用更多的专业协议来限制设计者的自由度?虽然现在标准网络协议已经广泛使用,但标准网络协议对高级交互如服务通信并不适用。

3. 可伸缩性。系统应如何构建才能可伸缩?也就是,整个系统如何来设计才能在外部的请求增加时增加处理能力?

4. 信息安全性。如何定义和实现可用的信息安全策略,并应用到一群独立管理的系统中?

5. 服务质量。交付给系统用户的服务质量应如何定义?系统应如何实现才能给所有的用户一个可接受的服务质量?

6. 失效管理。系统失效如何检测、缓解(使对系统的其他构件造成的影响最小)和修复?



通用对象请求代理体系结构 (Common Object Request Broker Architecture, CORBA) 是作为一个针对中间件系统的规格说明由对象管理组织 (OMG) 在 20 世纪 90 年代提出的。它的目的是作为一个开放标准来支持中间件的开发,从而实现分布式构件通信和执行并提供一组可以由这些构件使用的标准服务。

目前已经有了多个 CORBA 的实现,但是这些系统并没有得到广泛应用。用户倾向于使用专用的商业化系统 (例如 Microsoft 或 Oracle 提供的系统), 或者转向面向服务的体系结构。

<http://software-engineering-book.com/web/corba/>

理想情况下,系统是分布式的这一点对于用户应该是透明的。这意味着用户会把系统看成一个单一的系统,这个单一的系统行为不会受到系统分布方式的影响。然而,事实上这是不可能的,因为系统整体并没有集中控制。因此,一个系统中的各个构件在不同时候的行为可能各不相同。此外,由于信号在网络中的传输需要一定的时间长度,网络延迟是不可避免的。延迟的长度依赖于系统中资源的位置、用户网络连接的质量和网络负载。

为了使分布式系统透明(即隐藏其分布式特性),必须隐藏底层分布。你可以通过抽象隐藏系统资源,以使得这些资源的位置和实现可以在不影响分布式应用的情况下进行变化。中间件(将在 17.1.2 节中讨论)用于将程序中引用的逻辑资源映射到实际的物理资源上,并且管理这些资源之间的交互。

实际上,一个系统完全透明是不可能的,通常,用户会意识到他们正在与一个分布式系统打交道。因此,你可能会决定最好是向用户显示这是个分布式系统。那么,他们就可以为分布式系统所可能带来的后果(比如网络延迟、远程节点失效等)做准备。

开放性的分布式系统是依据普遍接受的标准来建立的系统。这意味着供应商提供的构件可以整合到系统中而且可以和其他系统构件互操作。在网络层面,开放性现在理所当然地被认为要遵从互联网协议,但是在构件层面,开放性还没有普及。开放性意味着系统构件可以使用任何编程语言独立开发,并且如果这些构件符合标准,将可以和其他构件一起运行。

20 世纪 90 年代 CORBA 标准(Pope, 1997)制定,旨在实现开放性,但是从未达到临界规模。相反,许多公司选择使用 Sun 公司(现在的 Oracle)和微软公司私有的构件标准来开发系统。这些标准提供了更好的软件实现和软件支持以及对工业协议更好的长期支持。

面向服务体系结构的 Web 服务标准(在第 18 章中讨论)有可能发展成为开放的标准。然而,这些标准因为效率低下而有很大的阻力。一些面向服务系统的开发者选择被称作 RESTful 的协议来作为替代品,因为这些协议本身要比 Web 服务协议的开销低很多。但 RESTful 协议也未标准化。

系统的可伸缩性反映了系统能在外部需求增加的情况下提供高质量的服务的能力。可伸缩性的三个维度是规模、分布和可管理性。

1. 规模。系统应该增加更多的系统资源来应对越来越多的用户。理想情况下,随着用户数量的增加,系统应该自动增加规模来处理用户数量的增加。

2. 分布。应当可以在不影响系统性能的情况下实现系统构件地理上的分散部署。当增加新构件时,应当不用关心它们的位置。大公司可以经常利用他们分布在世界各地不同设施上的计算资源。

3. 可管理性。即使系统的各部分位于独立组织中,也可以随着系统规模的增大而管理该系统。这是规模上最困难的挑战之一,因为它涉及管理人员沟通和商定管理政策。在实践中,系统的可管理性通常是制约系统可扩展程度的重要因素。

所谓的规模,有增强扩展(*scaling up*)和增加扩展(*scaling out*)的区别。增强扩展意味着用更强大的资源替换系统中的资源。例如,你或许会把服务器的内存由 16GB 增加到 64GB。增加扩展是指向系统增加更多的资源(例如,增加一个额外的服务器与现存的服务器一起工作)。增加扩展通常要比增强扩展更有成本效益,但是这意味着系统要设计得能并行处理才行。

在本书的第二部分中,已经讨论了一般的信息安全问题和信息安全工程的问题。但是,与集中式系统相比,分布式系统可能遭到攻击的方式就会明显增加。如果系统的一部分被成

功的攻击,那么攻击者很可能以此作为“后门”来入侵系统的其他部分。

分布式系统必须防御以免遭到以下类型的攻击:

1. 拦截,系统部件之间的通信被攻击者拦截,因此将会丧失机密性。
2. 中断,系统的服务遭到攻击并且不能按照预期来提供服务。拒绝服务攻击包括使用非法请求轰击一个节点,使得该节点不能处理合法的请求。
3. 更改,系统中的服务和数据被攻击者修改。
4. 伪造,攻击者生成本不应该存在的信息并且使用这些信息获得一些权限。例如,一个攻击者可能会生成一个不真实的密码入口并借此访问系统。

分布式系统最大的难点是建立一个能可靠地应用于系统中所有构件的信息安全策略。如第11章所述,一个信息安全策略规定了一个系统所能达到的安全级别。信息安全机制,比如加密和身份认证,被用来执行信息安全策略。在分布式系统中的难点是,由于不同的机构可能拥有各自一部分系统。这些机构或许有互不兼容的信息安全策略和信息安全机制。不得不制定安全协议以允许系统一起工作。

分布式系统提供的服务质量(QoS)反映了系统的一种能力,即可靠地提供服务并使得响应时间和吞吐量对于用户来说都是可接受的。理想情况下,服务质量需求应该事先明确定义并且设计和配置系统来提供这样的服务质量。不幸的是,这通常是不现实的,有以下两个原因。

1. 设计和配置系统提供高负载下的高服务质量是不符合成本效益的。高负载意味着需要更多的资源来确保合理的响应时间。云计算的出现使得这一问题得到了缓解,云服务器可以按照所需要的时长从云提供商那里租用。当外部请求增加时,可以自动增加更多的服务器。
2. 服务质量参数可能会相互矛盾。例如,可靠性的提升可能意味着吞吐量的减少,这是由于引进了检查程序来保证所有的系统输入都是合法的。

当系统处理时间敏感的数据(如语音或视频流)时,服务质量是至关重要的。在这种情况下,如果服务质量降到了阈值以下,那么语音或视频质量可能会下降到无法理解的地步。处理语音和视频的系统中应该包含负责服务质量协商和管理的构件。这些构件应当根据可用的资源对服务质量需求进行评估,如果需求无法满足,那么要协商获得更多的资源或者降低服务质量目标。

在分布式系统中出现失效是不可避免的,所以系统在设计上必须要适应这些失效。失效是无处不在的,以至于分布式系统的著名研究者 Leslie Lamport 对分布式系统提出了一个著名的定义:

只有当一个你从未听说过的系统崩溃阻止你做任何工作的时候,你才知道你所拥有的是一个分布式系统^①。

现在更真实的情况是,越来越多的系统在云中执行。失效管理包括应用容错技术(在第11章提到)。因此,分布式系统应当包含监测系统构件是否失效的机制,应当在失效发生时继续提供尽可能多的服务,应当尽量自动地从失效中恢复。云计算的一个重要优点是它大大降低了提供冗余系统构件的成本。

17.1.1 交互模型

分布式计算系统中的计算机之间可能会发生两种基本类型的交互:过程式交互和基于消息的交互。过程式交互指的是一台计算机调用其他计算机提供的一个已知的服务并等待服务

① Leslie Lamport, in Ross J. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems (2nd ed.), Wiley (April 14, 2008).

提供。基于消息的交互指的是“发送”计算机在消息中定义所需要的信息并发送给另一台计算机。较之对另一台机器的过程调用，消息总是在一个单独的交互中传送更多的信息。

为了说明过程式交互和基于消息的交互的区别，我们考虑在餐馆中点菜的情形。当你和服务员交谈时，你是在一系列同步的过程式交互中指定你要的东西。你做出了请求；服务员收到请求；你做出另一请求，请求被收到；等等。这好比软件系统中的构件交互，其中一个构件调用来自其他构件的方法。服务员记下你所点的菜，以及和你一起的其他人所点的菜，然后服务员将包括了所点的每个菜的细节的消息传递给厨房员工以准备食物。本质上讲，服务员传递消息给厨房员工，消息定义了需要准备的食物。这是基于消息的交互。

用餐者和服务员之间的过程式交互如图 17-1 所示，说明了如同一系列调用的同步点菜的过程，而图 17-2 给出了一个假设的 XML 消息，它定义了一桌 3 人所点的菜。这之间的信息交流方式的差异是很明显的。服务员所记录的是一系列的交互，每一个交互定义了所点的一部分菜。而服务员和厨房员工之间有一个单独交互，所传递的消息定义了完整的点菜内容。

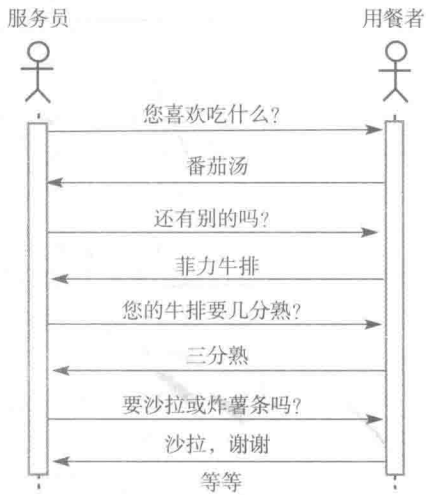


图 17-1 用餐者和服务员之间的过程式交互

```
<starter>
  <dish name = "soup" type = "tomato" />
  <dish name = "soup" type = "fish" />
  <dish name = "pigeon salad" />
</starter>
<main course>
  <dish name = "steak"-type = "sirloin" cooking = "medium" />
  <dish name = "steak" type = "fillet" cooking = "rare" />
  <dish name = "sea bass">
</main>
<accompaniment>
  <dish name = "french fries" portions = "2" />
  <dish name = "salad" portions = "1" />
</accompaniment>
```

图 17-2 服务员和厨房员工之间的基于消息的交互

在分布式系统中过程式通信往往是通过远程过程调用（remote procedure call, RPC）实现的。在远程过程调用中，一个构件调用另一个构件就像是调用本地的程序或方法。系统中的中间件拦截调用并把它传送给一个远程的构件。远程构件执行了所要的计算，通过中间件返回结果给调用构件。Java 中，远程方法调用（remote method invocation, RMI）与远程过程调用相似，但不完全相同。RMI 框架在 Java 程序中处理对远程方法的调用。

RPC 要求一个“桩”以便所要调用的过程在发起请求的计算机上是可访问的。桩定义了远程过程的接口，这个桩被调用，并把过程的参数转化为一种标准的表示以便传送给远程

过程。接着，桩通过中间件发送执行请求给远程过程。远程过程使用库函数将参数转换成需要的格式，执行相应计算，并通过代表调用方的“桩”返回结果。

基于消息的交互通常是构件创建一个消息，这个消息详细地说明了来自另一个构件的服务请求。通过中间件，消息发送给接收构件。接收者分析消息，执行计算，并为发送构件创建一个带有所需要的结果的消息。这个消息接着传到中间件以便传送给发送构件。

远程过程调用方法带来的一个问题是调用者和被调用者需要在通信时都是有效的，它们必须知道如何相互指引。本质上，远程过程调用和本地过程和方法调用有着同样的需求。相反，在基于消息的方法中，不可用是可以容忍的，因为消息只需要等在队列里直到接收者变为可用。此外，消息的发送者和接收者没必要知道彼此。他们只是与中间件通信，中间件负责确定消息传送到合适的系统中。

17.1.2 中间件

在分布式系统中，不同的构件可能用不同的程序语言来实现，且这些构件可能运行在不同类型的处理器上。数据模型、信息表示法以及通信协议可能都不一样。因此，分布式系统就需要某种软件来管理这些不同部分，确保它们能通信和交换数据。

中间件这个术语指的就是这样一种软件，它位于系统的不同分布式构件之间。图 17-3 说明了中间件是操作系统和应用程序中间的一层。中间件通常实现为一系列库，这些库被安装到每一个分布式计算机上，加上一个运行时系统管理通信。

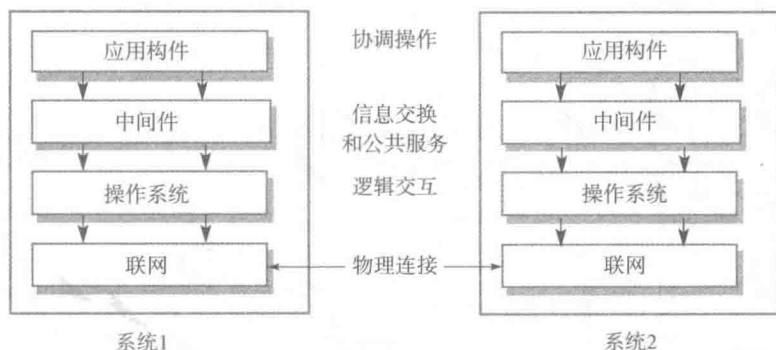


图 17-3 分布式系统中的中间件

Bernstein(1996)总结了支持分布式计算的中间件的不同类型。中间件是一种通用软件，通常中间件不是由应用开发人员编写的，而是购买成品软件系统。中间件的例子有负责数据库通信管理的软件、事务管理器、数据转换器和通信控制器等。

在分布式系统中，中间件通常提供两种不同类型的支持。

1. 交互支持，中间件协调系统中的不同构件之间的交互。中间件提供位置透明性，因为构件不需要知道其他构件的物理位置。如果使用不同的编程语言来实现构件、事件检测和通信等时，那么中间件还可能会支持参数转换。

2. 提供公共服务，即中间件提供对服务的可复用的实现，分布式系统中的很多构件可能需要这些服务。通过使用公共服务，构件可以很容易地相互协作，并且可以持续地向用户提供服务。

17.1.1 节已经给出了中间件可以提供的交互支持的例子。你可以使用中间件来支持远程

过程调用、远程方法调用和消息交换等。

公共服务是指各种不同的构件都需要的服务，不管这些构件的功能是什么。如第 16 章中所述，这些服务可能包括信息安全服务（身份认证和权限检查）、通知和命名服务，以及事务管理服务。对于分布式构件，你可以把这些服务看作是中间件容器提供的；对于服务，则是通过共享库提供的。你可以部署你的构件，并且这些构件可以访问和使用这些公共服务。

17.2 客户 - 服务器计算

通过互联网访问的分布式系统通常组织成客户 - 服务器系统。在客户 - 服务器系统中，用户与运行在本地计算机上的程序（例如，Web 浏览器或移动设备上的 APP）交互。这个程序与运行在远程计算机上（例如 Web 服务器）的另一个程序交互。远程计算机提供如网页访问等服务，这些服务对于外部客户端都是可用的。如第 6 章中所述，这种客户 - 服务器模型是一个应用程序非常普遍的体系结构模型。这种模型不受分布在许多计算机上的应用程序的限制。你也可以使用这种模型作为一种逻辑交互模型，即让客户端和服务端运行在同一台计算机上。

在客户 - 服务器体系结构中，将应用建模为由服务器所提供的一系列服务。客户端可以访问这些服务并提交结果给最终的用户。客户端需要知道可用服务器的存在但不知道其他客户端的存在。客户端和服务端是独立的进程，如图 17-4 所示。该图说明了这样一种情况，有 4 台服务器（s1 ~ s4）分别提供不同的服务，每一台服务器都有一组与之关联的客户端来访问这些服务。

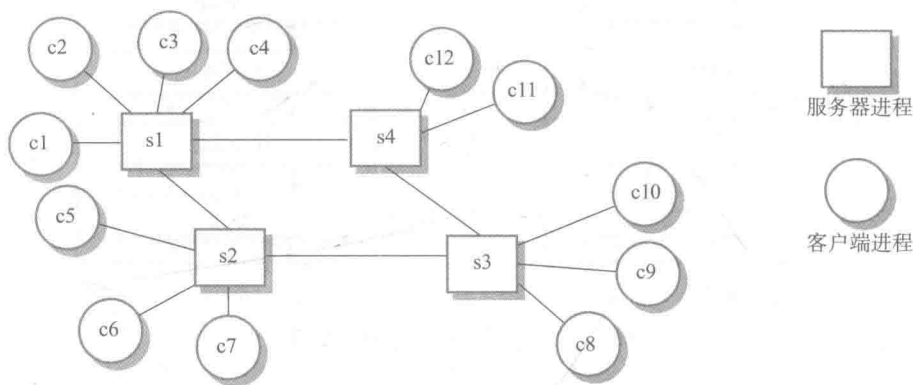


图 17-4 客户 - 服务器交互

图 17-4 示意的是客户端和服务端进程而不是处理器。正常情况下，在单个处理器上会运行多个客户端进程。例如，在你的 PC 上，你可能会运行邮件客户端从远程邮件服务器上下载邮件。你也可能会运行一个 Web 浏览器与远程 Web 服务器交互，你或者运行一个打印客户端程序向远程打印机发送文件。图 17-5 说明了有 12 个逻辑的客户端（如图 17-4 所示）运行在 6 台计算机上的情况。4 个服务器进程被映射到两台物理的服务器计算机上。

多个不同的服务器进程可以运行在同一个处理器上，但是服务器经常被实现为多处理器系统，其中每台机器上运行一个单独的服务器进程实例。负载均衡软件把客户端向服务器发送的请求分配给不同的服务器，以使得每一个服务器承担同样的工作量。这就使更多的与客户端的交互事务得以处理，而不会降低对单个客户端的响应。

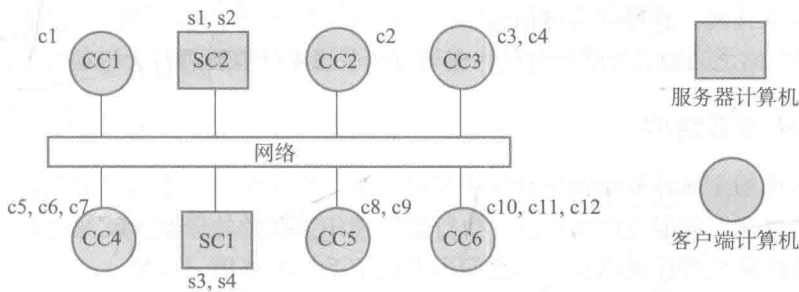


图 17-5 客户端和服务映射到网络计算机上

客户 - 服务器应用的分层体系结构模型中，客户 - 服务器系统清晰地分离了信息表示和信息计算，信息计算是对信息的创建和处理。因此，应该设计出分布式客户 - 服务器系统的体系结构，让信息的表示和信息计算分布在几个逻辑的分层上，层与层之间有明确的接口。这种设计允许每一层分布在不同的计算机上。图 17-6 说明了这种模型，它将一个应用分解为 4 层。

1. 表示层，是关于呈现信息给用户以及管理所有用户的交互。
2. 数据管理层，用来管理传给客户端和来自客户端的数据。这一层可以实现数据检查、网页生成等。
3. 应用处理层，涉及应用逻辑的实现以及向最终用户提供所需的功能。
4. 数据库层，用来储存数据和提供事务管理服务。

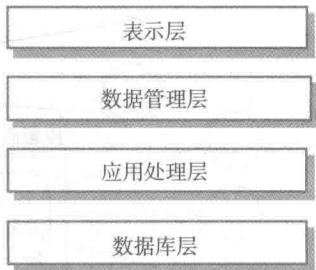


图 17-6 客户 - 服务器应用的分层体系结构模型

接下来的章节解释不同的客户 - 服务器体系结构如何以不同的方式分布这些逻辑的分层。客户 - 服务器模型也强调了软件即服务（Software as a Service, SaaS）的概念，这一概念已成为部署软件和通过互联网访问软件的一种越来越重要的方法。17.4 节将会讨论这一问题。

17.3 分布式系统的体系结构模式

如本章的引言中所述，分布式系统的设计者必须要组织他们的设计以便在系统的性能、可依赖性、信息安全性和可管理性之间找到平衡。因为没有适用于所有环境的一种通用的系统组织模型，所以出现了不同的分布式体系结构风格。当设计一个分布式应用时，应该选择一种能支持你的系统中关键的非功能性需求的体系结构风格。

本节将会介绍以下 5 种体系结构风格。

1. 主从体系结构，这种体系结构常用在实时系统中，在这类系统中交互的响应时间需要得到保证。
2. 两层客户 - 服务器体系结构，这种体系结构常用于简单的客户 - 服务器系统，也用于出于信息安全原因，需要使系统集中化的情况。
3. 多层客户 - 服务器体系结构，这种体系结构用于服务器要处理大量的事务的时候。
4. 分布式构件体系结构，这种体系结构用于来自不同系统和数据库的资源需要结合在一起的时候，或者作为多层客户 - 服务器系统的实现模型。

5. 对等体系结构, 这种体系结构用于当客户端相互交换本地存储的信息并且服务器的角色仅仅是在客户端之间相互介绍的时候, 也可以用于当系统需要进行大量独立的计算的时候。

17.3.1 主从体系结构

分布式系统的主从体系结构通常用于实时系统。在这类系统中, 可能存在分别与获取来自系统环境的数据、数据处理和计算、执行器管理相关联的处理器。就像第 21 章将讨论的, 执行器是由软件系统所控制的设备, 它们用来改变系统的环境。例如, 一个执行器可能控制一个阀门并把阀门的状态由“开”变为“关”。“主”进程通常负责计算、协调和通信, 并控制“从”进程。“从”进程用于执行专门的行为, 比如从传感器阵列中获取数据。

图 17-7 说明了这种体系结构模型。这是一个城市交通管理系统的模型, 这个模型有 3 个运行在不同处理器上的逻辑进程。主进程是控制室进程, 它与负责收集交通数据和管理交通信号灯的独立的从进程通信。

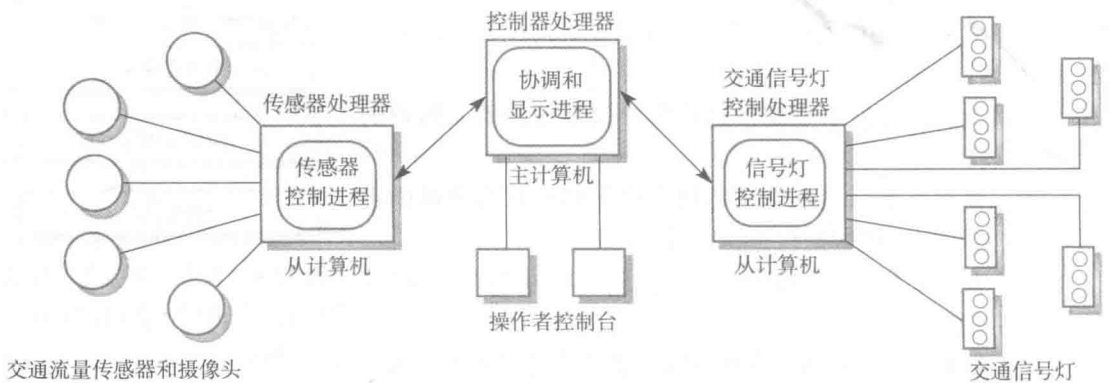


图 17-7 交通管理系统的主从体系结构

一组分布式传感器收集交通流量信息。传感器控制进程周期性地轮询传感器来获取交通流量信息并整理这些信息以便进一步处理。传感器处理器本身也被主进程周期性地轮询来提供信息。主进程负责向操作者显示交通状况, 计算交通信号灯序列, 接受操作者的指令来修改这些序列。控制室系统发送指令给交通信号灯控制进程, 交通信号灯控制进程把这些命令转化为信号来控制交通信号灯硬件。主控制室系统本身组织为一个客户-服务器系统, 客户端进程运行在操作者控制台上。

能够预测所需要的分布式处理和可以很容易地将处理定位到从处理器上的这两种情形中, 我们可以使用分布式系统的主从模型。这种情况一般发生在实时系统中, 在实时系统中最重要的是要满足处理的时限。可以使用从处理器于计算密集型操作, 比如信号处理和系统控制的设备管理。

17.3.2 两层客户-服务器体系结构

在 17.2 节中讨论了客户-服务器系统的一般形式, 在客户-服务器系统中一部分应用系统运行在用户的计算机上(客户端), 另一部分运行在远程计算机上(服务器)。此外, 前文还提出了一种分层应用模型(见图 17-6), 其中系统中的不同层可能会在不同的计算机上执行。

两层客户-服务器体系结构是客户-服务器体系结构最简单的形式。系统实现为一个独立的逻辑服务器加上不确定数目的使用服务器的客户端。图 17-8 显示了这种体系结构模型的两形式。

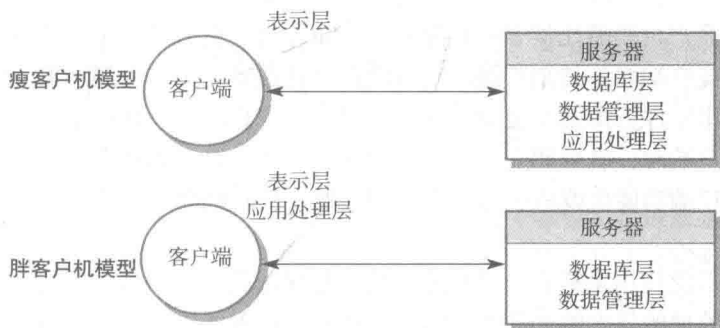


图 17-8 瘦客户机和胖客户机的体系结构模型

1. 瘦客户机模型，其表示层在客户端实现，其他层（数据管理层、应用处理层和数据库层）在服务器上实现。客户端上的表示层软件通常是通过 Web 浏览器，但在移动设备上也可以使用应用程序。

2. 胖客户机模型，一部分或者所有的应用处理都在客户端上执行。数据管理和数据库功能在服务器上实现。在这种情况下，客户端软件可以是专门编写与服务器应用程序紧密集成的程序。

瘦客户机模型的优点是管理客户端很简单。如果存在很多的客户端那么管理就是最主要的问题，因为在所有的客户端上安装新的软件是非常困难和昂贵的。如果使用 Web 浏览器作为客户端，就不需要安装任何软件。

瘦客户机模型的缺点是它将繁重的处理负载都放在了服务器和网络上。服务器负责所有的计算，这将导致客户端和服务器之间的网络上很大的流量。使用这种模型来实现一个系统可能需要在网络和服务容量上投入更多。

胖客户机模型利用运行客户端软件的计算机上的处理能力，并把部分或所有的应用处理和表示分发到客户端上。实际上，服务器就是一个事务服务器，即管理所有的数据库事务。数据管理是简单的，因为不需要管理客户端和应用处理系统之间的交互。当然，胖客户机模型的问题是它需要额外的系统管理来部署和维护客户端上的软件。

使用胖客户机体系结构的一个例子是银行的 ATM 系统，ATM 系统向用户提供现金和其他的银行服务。这里 ATM 是一个客户端，服务器是一个典型的大型主机，在它上面运行客户账户数据库。大型主机是一个为事务处理而设计的强大的机器。因此这个大型主机可以处理由 ATM 机、其他的柜台系统和在线银行产生的大量事务。自动取款机上的软件执行很多与用户相关的处理，这些处理与交易有关。

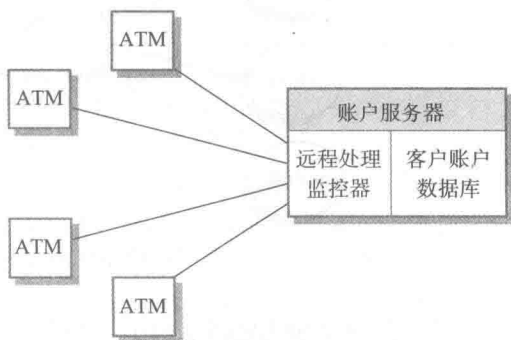


图 17-9 ATM 系统的胖客户机体系结构

图 17-9 显示了 ATM 系统组成的一个简化

版本。注意, ATM 不是直接连接到客户账户数据库上, 而是连到一个远程处理 (teleprocessing, TP) 监控器上。远程处理监控器是一个中间件系统, 它组织远程客户端和数据库间的通信并序列化客户端交易以便数据库处理。这确保了交易的独立并且不会干涉其他交易。使用事务序列化的好处是系统能从故障中恢复而不至于造成系统数据崩溃。

一方面, 胖客户机模型比瘦客户机模型在分布处理上更有效; 另一方面, 如果面对一个具有特定目的的客户端而不是浏览器, 它将使系统管理更复杂。应用功能分散在许多不同的计算机上。当应用软件不得不变更时, 就需要对每个客户端计算机进行重新安装。当系统中有数以百计的客户端时, 就可能造成比较大的成本开销。自动更新客户端软件可以降低这些成本, 但如果客户端功能更改将引入它自己的问题。新功能可能意味着企业必须改变他们使用系统的方式。

移动设备的广泛使用意味着, 尽可能地最小化网络流量是非常重要的。这些设备现在包含可以进行本地处理的强大的计算机。因此, 瘦客户机和胖客户机体系结构之间的区别已经变得模糊。应用程序可以具有能执行本地处理的内置功能, 并且网页可以包含在用户本地计算机上执行的 Javascript 构件。应用程序的更新问题仍然是个难题, 但通过自动更新应用程序而非显式的用户干预, 它已在一定程度上得到了解决。因此, 使用这些模型作为分布式系统体系结构的一般基础, 虽然有时是有帮助的, 但在实践中很少有基于 Web 的应用程序在远程服务器上实现所有的处理。

17.3.3 多层客户 - 服务器体系结构

两层客户端服务器方法的根本问题是系统中的逻辑层 (表示层、应用处理层、数据管理层和数据库层) 必须要映射到两个计算机系统上: 客户端和服务端。如果选择的是瘦客户机模型, 则可能导致可伸缩性和性能的问题; 若选择的是胖客户机模型, 则可能有系统管理上的问题。为了避免其中的一些问题, 可以使用一种“多层客户 - 服务器”的体系结构。在这种体系结构中, 系统中的不同层 (表示层、数据管理层、应用处理层和数据库层) 是在不同处理器上执行的独立的进程。

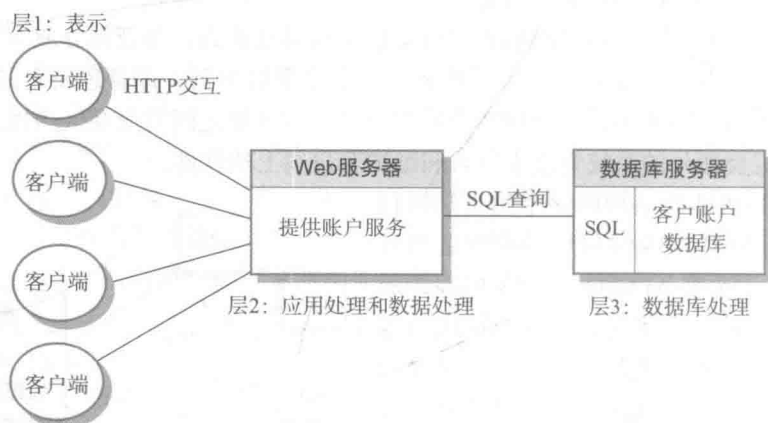


图 17-10 网上银行系统的三层体系结构

网上银行系统 (见图 17-10) 是使用多层客户 - 服务器体系结构的例子, 这个系统中有 3 层。银行的客户数据库 (通常在如上所述的一个大型计算机上) 提供数据库服务。一个

Web 服务器提供（如网页的生成）数据管理服务和一些应用服务。诸如现金转账、生成银行结算单和工资单等应用服务在 Web 服务器上实现，并作为脚本在客户端上执行。用户计算机和浏览器构成客户端。这种系统是可扩展的，因为当客户数量增加的时候增加服务器（增加扩展）相对还是比较容易的。

在这种情况下，使用三层体系结构可以对 Web 服务器和数据库服务器之间信息传递进行优化。这两个系统之间的通信无须基于互联网标准，可以使用比较低层的数据交换协议。使用支持结构化查询语言（SQL）数据库查询的高效中间件从数据库取回信息。

在系统中添加更多的服务器，可以把三层的客户－服务器模型扩展成多层结构。这可能包括使用 Web 服务器管理数据，使用独立的服务器处理应用程序以及数据库服务。多层系统适合于当应用程序需要到不同的数据库中存取数据的情况。在这种情况下，或许需要在系统中增加一个集成的服务器。这个集成服务器收集分布式数据并将其传递给应用程序，就好像数据是在一个单一的数据库中一样。如下一节中将要提到的，分布构件体系结构可以用来实现多层客户及服务器系统。

在多个服务器上分布应用处理的多层客户－服务器系统，本质上比两层的体系结构有更强扩展性。系统中的层可以独立管理，随着负载增加而添加额外的服务器。处理可以分布到不同的应用逻辑服务器和数据管理服务中，以便谋求更快的客户响应。

当选择最适当的分布式体系结构时，客户－服务器体系结构的设计者必须考虑到多种因素。客户－服务器体系结构的适应条件如图 17-11 所示。

体系结构	应 用
两层瘦客户机 C/S 体系结构	遗留系统应用，当将应用处理过程和数据管理分离是不切合实际的时候使用。客户端将它们作为服务访问，如在 17.4 节中所讨论的那样 计算密集型应用，例如编译器，很少或根本没有数据管理 很少或根本没有密集型应用处理的数据密集型应用（浏览和查询）。浏览 Web 页面是使用此体系结构情形的最为普通的例子
两层胖客户机 C/S 体系结构	其应用处理是在客户端上由 COTS（如微软的 Excel）所提供的这类应用 需要密集的数据处理（如数据可视化）的应用 不能保证互联网连接的移动应用。因而，某些使用来自数据库的缓存信息的本地处理是可能的
多层 C/S 体系结构	大规模的、具有成百上千个客户端的应用 数据和应用都是易变的一类应用 多源数据经过集成而得的一类应用

图 17-11 客户－服务器（C/S）体系结构模式的应用

17.3.4 分布式构件体系结构

如图 17-6 所示，通过把进程组织到各个层中去，系统的每一层可以实现为一个独立的逻辑服务器。这个模型对许多类型的应用来说能工作得很好。然而，它对系统设计者的灵活性是一个很大的限制，因为设计者必须决定每一层应该包括哪些服务。然而在实际中，一个服务到底是数据管理服务、应用服务还是数据库服务不总是很明确。设计者还必须考虑可伸缩性，从而为当更多的客户端被加入系统中时的服务器复制提供一些方法。

分布式系统设计的更通用方法是把系统设计为一系列服务，不需要试图把这些服务定位到系统中的某一层上。每一个服务或者相关的一组服务是使用独立的构件实现的。如第 16 章所述，在分布式构件体系结构中（见图 17-12），系统被组织成一系列交互的构件或对象。这些构件给出了它们所提供服务的接口。其他构件通过中间件使用远程过程调用或远程方法调用请求这些服务。

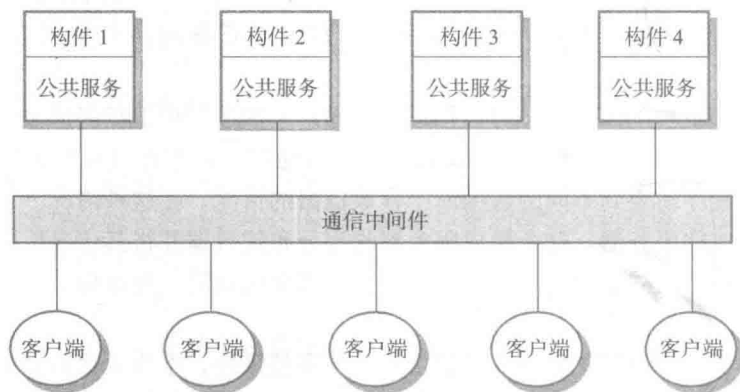


图 17-12 分布式构件体系结构

分布式构件系统依赖于中间件来管理构件间的交互，弥合构件间传递的不同类型参数之间的差异，并提供一系列公共服务供应用构件使用。CORBA 标准（Orfali, Harkey, and Edwards 1997）是这种中间件早期的一个例子，但是现在应用得不是很广泛了。它已在很大程度上被专有的软件所取代，比如 EJB（Enterprise Java Beans）或者 .NET。

使用分布式构件模型来实现分布式系统具有以下好处。

1. 它允许系统设计者延迟决定在哪里和如何提供服务。提供服务的构件可以在网络任何节点上运行。没有必要事先决定一个服务是数据管理层的一部分，还是应用层的一部分，抑或用户接口层的一部分。
2. 它是一个非常开放的体系结构，允许新的资源根据需要增加进来。可以很容易地添加新的系统服务而不会对现有的系统产生很大的影响。
3. 系统具有很好的灵活性和可伸缩性。当系统负载增加时，可以增加新的对象或者重复的对象，而不中断系统的其他部分。
4. 通过构件在网络上的迁移达到对系统的动态配置是有可能的。这在服务需求模式不确定的场合中可能是很重要的。一个提供服务的构件能迁移到请求服务的构件所在的同一台处理器上，这样可以改善系统的性能。

分布式构件体系结构可以作为一个逻辑模型来构造和组织系统。在这种情况下，要考虑该如何提供应用功能，把这些功能按照服务或服务组合的形式给出。然后设计如何利用分布式构件来提供这些服务。举例来说，在一个零售应用中可能有关于库存控制、客户通信、货物订购等应用构件。

数据挖掘系统是这种系统的一个很好的例子，在数据挖掘系统中分布式构件体系结构是最合适使用的体系结构。数据挖掘系统寻找在多个不同数据库中数据间的关系（见图 17-13）。数据挖掘系统通常从许多独立的数据库中得出信息，执行计算密集型处理，并将发现的关系用易于理解的形式显示出来。

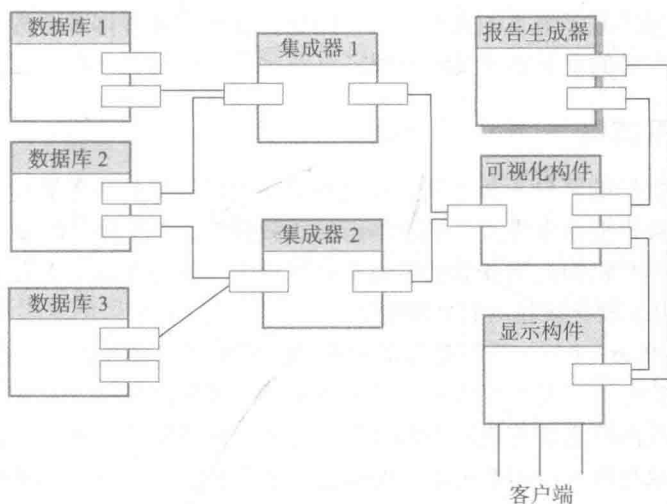


图 17-13 数据挖掘系统的分布式构件体系结构

这种数据挖掘应用的一个例子可以是一个出售食物和书籍的零售业系统。零售企业使用一个单独的数据库来维护关于食品和书籍的详细信息。他们使用会员卡系统来跟踪客户的购物活动，因此有一个大型数据库来链接产品的条形码和客户信息。市场部门想要寻找客户购买食物和书籍之间的联系。例如，相当多的人在买比萨的同时也会购买犯罪小说。因此，企业可以在新小说出版的时候明确地向购买具体食物的客户提供新小说的信息。

在这个例子中，每个数据库可以封装成一个分布式构件，其接口提供只读访问功能。每个集成器构件专注于一种特定类型的关系，它们从所有的数据库中收集信息来试着推导出这种关系。例如，可能有一个集成器构件关注食品销售的季节性变化关系，还有一个关注不同类型商品之间的销售关系。

可视化构件通过与集成器构件交互来给出所发现的关系的可视化效果或报告。由于所要处理的数据量巨大，可视化构件通常通过图形化的方式展示它们的结果。最后，显示构件可以负责向客户端传递图形化模型以便在 Web 浏览器上呈现最终结果。

对这一类型应用，分布式构件体系结构比客户 - 服务器体系结构更适合，因为你可以在系统中增加新的数据库而不会产生很大的影响。每一个新的数据库只是加入另一个访问分布式构件。数据库访问构件提供一个简化的接口以控制对数据的访问。被访问的数据库可以存在于不同的机器上。该体系结构也使得通过添加新的集成构件来挖掘新的关系类型变得简单。

分布式构件体系结构有以下两个主要缺点。

1. 其设计比客户 - 服务器系统更复杂。多层客户 - 服务器系统看起来是一种相当直观的思考系统的方式。它反映了很多人类事务的处理方式，其中人们从其他专门提供这些服务的人那里请求和接收服务。分布式构件体系结构的复杂性增加了实现的成本。

2. 分布式构件模型或中间件没有通用标准。不同的提供商（比如微软和 Sun 公司）开发出了各种相互不兼容的中间件。这种中间件非常复杂，依赖它们会增加整个分布构件系统的复杂度。

因为这些问题，在很多情况下，分布式构件体系结构都被面向服务的系统（在第 18 章讨论）取而代之。不过，分布式构件系统要比面向服务的系统性能更优越。RPC 通信通常要

比面向服务系统使用的基于消息的交互更快。因此，分布式构件体系结构更适合高吞吐量的系统，这种系统中大量的事务需要及时处理。

17.3.5 对等体系结构

本章的前几节提到的客户-服务器计算模型对于提供服务的服务器和接收服务的客户端有明确的区分。这种模型通常导致系统不均衡的负载分布，其中服务器要比客户端做更多的工作。这就会导致组织机构在服务器的能力上花费巨大，然而在成千上万的用来访问系统服务器的 PC 上却有很多没有被利用的处理能力。

对等 (Peer-to-Peer, P2P) 系统是分散式系统，它的计算是由网络上的任意一个节点来承担的，至少从理论上讲，是不存在客户端和服务端之间的区别的。对于对等式应用，总的系统设计建立在潜在的巨大计算机网络的计算能力和存储资源的优势上。保证节点间能有效通信的标准和协议都是嵌入应用系统本身的，每个节点必须运行一个这样的应用的拷贝。

对等技术主要用于个人而不是商业系统。事实上，没有中央服务器意味着这些系统更难以监控；因此，有可能实现更高级别的通信隐私。

例如，我们使用基于 BitTorrent 协议的文件交换系统在个人计算机上共享文件，又例如，我们使用即时消息系统（如 ICQ 和 Jabber）在两个用户之间通信而无须中间服务器提供支持。比特币是使用比特币电子货币的 P2P 支付系统。Freenet 是一个去中心化的数据库，设计用于匿名的信息发布，避免权力机构对这些信息的查禁。

已经开发出的其他一些对等系统，其中隐私不是主要需求。利用 IP 的语音 (VoIP) 电话服务，比如 Viber，依赖于电话呼叫或会议的相关各方之间的对等通信。SETI@home 是一个长期项目，它在家庭个人计算机上处理来自射电望远镜传来的数据，搜索宇宙中生命的迹象。在这些系统中，P2P 系统模型的优势是，中央服务器不再是运算瓶颈。

然而，企业也利用 P2P 系统来更好地使用它们个人计算网络的能力 (McDougall 2000)。Intel 公司和 Boeing 公司都实现了 P2P 系统用于计算密集型应用。这样就利用了本地计算机没有使用的处理能力。工程计算可以整夜地运行在没有使用的桌面电脑上，而不必购买昂贵的高性能硬件。企业也在广泛地使用商务 P2P 系统，比如消息系统和 VoIP 系统。

原则上，在对等系统中网络上的每一个节点都能够看到每个其他节点，可以与之建立连接，并可以与之直接交换数据。当然，实际过程中这是不可能的，所以节点都是以区域来组织的，通过某些节点作为桥梁来与其他区域中的节点群建立连接。图 17-14 展示了这种分散式 P2P 体系结构。

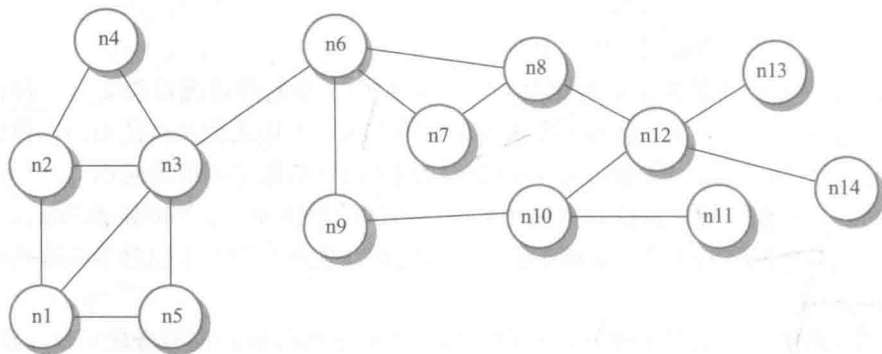


图 17-14 分散式的 P2P 体系结构

在分散式体系结构中,网络中的节点不仅仅是一个功能单元,它还是通信的转换器,能够从一个节点到另一个节点路由数据和控制信号。例如,假设图 17-14 代表一个分散式文件管理系统。该系统由一个研究人员团体来共享文件,每一个该团体中的成员维护他自己的文件库。然而,当一个文件被检索到,这个检索文件的节点还要使之对所有节点都是可用的。

如果有谁需要一个储存在网络中某个地方的一个文件,就发出一个搜索命令给本区域中的节点。这些节点检查是否它们有所要的文件,如果有,则将信息发回给请求者,如果它们没有,它们就将此搜索路由到其他节点。因此,如果 n1 发出对某个存储于节点 n10 上的文件的搜索命令,该搜索路由经过节点 n3、n6 和 n9,到达 n10。当文件最终得以发现,持有这个文件的节点通过对等连接把文件直接发送给请求节点。

这种分散式体系结构的优点在于它是高度冗余的,所以它是容错的而且容许节点从网络中断开。然而,系统的缺点是相同的搜索会被很多不同的节点来处理,这种重复的对等通信开销很大。

半集中式 P2P 体系结构是一种替代的 P2P 体系结构模型,是不同于单纯 P2P 体系结构的半集中式体系结构,即在网络中,有一个或多个节点充当服务器来提供对其他节点的通信。这减少了节点之间的流量,图 17-15 说明了半集中式体系结构模型和完全分散式模型(如图 17-14)的区别。

在半集中式体系结构中,服务器(有时叫作超级对等体)的作用是帮助建立网络中两个节点之间的联系,或者是协调计算的结果。例如,如果图 17-15 代表一个即时消息系统,那么网络节点与服务器通信(用虚线表示)来寻找哪些节点是可行的。一旦发现了它们,就能够建立它们之间的直接通信而与服务器的连接就不再需要了。因而,节点 n2、n3、n5 和 n6 处于直接通信状态。

在 P2P 计算系统中,把处理器密集的计算分布到很多个节点上去,通常是要有一些节点作为超级对等体,它们的作用是将工作分发到各个其他节点上并收集和检查计算的结果。

在以下两种环境下系统适合使用对等体系结构模型。

1. 系统是计算密集的,并且有可能把需求的处理分成许多独立的计算。例如,支持药物探索计算的对等系统分散了通过分析大量的分子来寻找治疗癌症潜在方法的计算,以此来观察这些分子是否具有所需要的抑制癌症增长的特性。每一个分子都可以独立地分析,所以系统中的对等体不需要通信。

2. 系统主要涉及个人计算机在网络上的信息交换,没有必要将这些信息集中管理和储存。这种应用的例子包括:允许对等体交换比如音乐和视频等本地文件的文件共享系统;支持计算机间语音和视频通信的电话系统。

对等体系结构能够有效利用网络能力。然而,信息安全问题是这些系统没有被更广泛使用的主要原因,尤其是在商业上(Wallach 2003)。缺乏集中式管理意味着攻击者可以设置恶意节点,传输垃圾邮件和恶意软件给合法的 P2P 系统用户。对等通信需要开放你的电脑直接与其他对等体交互,这意味着这些系统有可能访问你的任何资源。为了应对这种情况,

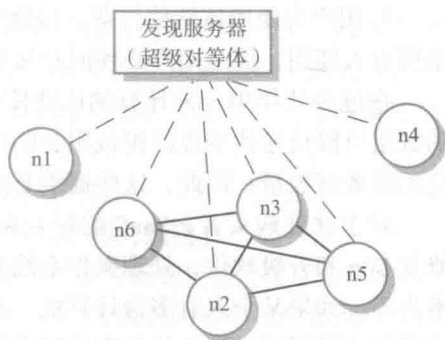


图 17-15 半集中式 P2P 体系结构

需要组织你的系统来保护这些资源。如果没有做好这些，你的系统就会不安全，容易受到外部攻击。

17.4 软件即服务

前几节讨论了客户-服务器模型以及功能是如何在客户端和服务端之间分布的。为了实现一个客户-服务器系统，可能需要在客户端上安装程序或 APP，与服务端通信，实现客户端功能并管理用户界面。例如，一个像 Outlook 或 Mac Mail 的邮件客户端，在自己的计算机上提供邮件管理的特征。这避免了瘦客户机系统中所有的进程都运行在服务端的过载问题。

不过，通过使用诸如 AJAX (Holdener, 2008) 和 HTML5 (Sarris 2013) 这样的 Web 技术，服务端超负载的问题得到了明显的改善。这些技术支持网页表示的有效管理和通过脚本语言的本地计算。这意味着浏览器可以作为客户端配置和使用，有很强的本地处理能力。应用软件可以被看作远程服务，可以被任何运行标准浏览器的设备访问。广泛使用的软件即服务 (Software as a Service, SaaS) 示例包括基于 Web 的邮件系统，例如 Yahoo 和 Gmail，以及办公应用程序，例如 Google Docs 和 Office 365。

SaaS 这个概念涉及远程托管软件以及通过互联网提供对软件的访问。SaaS 的关键要素如下。

1. 软件部署在一台服务器上 (更一般的情形是在云上)，并且可以通过 Web 浏览器访问。软件不是部署在本地 PC 上的。
2. 软件由软件提供商拥有和管理，而不是由使用软件的机构来管理。
3. 用户为使用该软件付费，根据他们的使用量包年或者包月来支付。有时，该软件免费给所有人使用，但是用户必须同意接受广告，因为广告为软件服务提供资金。

在过去几年中，云计算的广泛使用加速了 SaaS 的发展。当服务端部署在云上时，服务器的数量可以快速改变以匹配该服务的用户需求。没有必要一直为服务提供商提供峰值负载对应的服务器数量；因此，这些提供商的成本已大大降低。

对于软件购买者，SaaS 的好处在于软件管理的开销转移给了提供商。软件提供商负责修复 bug 和升级软件，处理操作系统平台的变化，确保硬件性能满足需求。软件许可管理成本为零。如果某个人有多台计算机，不需要为所有的计算机注册软件。如果一个软件应用只是偶尔才被用到，那么按每次使用支付要比购买这个应用更便宜。比如手机这种移动设备可以在世界的各个地方访问软件。

与远程服务的数据传输是抑制 SaaS 使用的主要问题。在网络传输速度慢时，传输大流量数据 (如视频或者高品质的图片) 要花费大量的时间。你或许还要根据传输量向服务提供商支付费用。其他问题包括缺少对软件演化的控制 (提供商可能在他们觉得合适的时候来更新软件)，以及法律法规的问题。许多国家有专门的法律来规定数据的存储、管理、维护和访问，移动数据到远程的服务可能会触犯这些法律。

SaaS 的概念和第 18 章要讨论的面向服务的体系结构 (SOA) 显然是很相关的，但又不一样。

1. SaaS 是在远程服务器上提供功能而客户端通过 Web 浏览器访问的一种方法。服务器在交互会话期间维持用户的数据和状态。事务常常是长事务 (例如，编辑文件)。
2. SOA 是把软件系统构建为一系列单独的无状态服务的方法。这些服务或许由多个提供商提供并且可能是分布的。典型地，事务是短事务，其中服务被调用，做一些处理，接着返回结果。

SaaS 是向用户交付应用功能的方法，而 SOA 是应用系统的一种实现技术。使用 SOA

实现的系统不一定要作为 Web 服务被用户访问。关于业务的 SaaS 应用程序可以使用构件而不是服务来实现。不过, 如果 SaaS 是使用 SOA 实现的, 那么应用程序就可能使用服务 API 来访问其他应用程序的功能。它们然后可以整合到一个更复杂的系统。这叫作混搭 (mashup), 代表了软件复用和快速软件开发的另一种方法。

从软件开发的角度来看, 服务的开发过程与其他的软件开发有很多的相似之处。不过, 服务的构建通常不是由客户的需求驱动的, 而是服务提供商基于对用户需求的假设来驱动的。因此软件需要能够在得到用户的需求反馈后很快地演化。因此, 敏捷开发和增量式交付是开发作为服务来部署的软件的一种常用方法。

一些实现为服务的软件, 例如网络用户使用的 Google Docs, 为所有用户提供了一种统一的体验。但是, 企业可能希望根据他们自己的需求定制特定的服务。当要实现 SaaS 的时候必须要考虑可能会有许多不同机构的软件用户。必须考虑以下 3 个重点要素。

1. 可配置性。如何针对每一个客户机构的特定需求对软件进行配置?
2. 多租户。如何让每一个用户感到他们正在使用他们自己的系统副本, 而同时能高效地使用系统资源?
3. 可伸缩性。如何设计系统以便能扩展来容纳不可预测的大量用户。

第 16 章介绍的产品线体系结构的思想, 是配置系统满足那些具有重叠但不完全相同需求的用户的一种方法。基于这种方法, 你可以在一个通用的系统基础上根据每个用户的特定需求对系统进行调整。

不过, 这对于 SaaS 是不起作用的, 因为这将意味着为使用这种软件的每个机构部署服务的一个不同副本。而且, 需要在系统中设计可配置性, 并提供一个配置接口以便允许用户指定他们的偏好。当软件使用的时候, 利用这些偏好来动态地调整软件的行为。配置设施可能会允许以下内容:

1. 品牌化, 为每一个机构的用户提供一个反映他们机构的接口。
2. 业务规则和工作流, 每个机构定义自己的服务和数据使用的规则。
3. 数据库扩展, 每个机构定义如何将通用服务数据模型扩展为能满足特定需求的模型。
4. 访问控制, 服务的客户为他们的员工建立个人账户, 并为他们的每个用户定制可访问的资源 and 功能。

图 17-16 说明了这种情况, 图中显示了应用服务的 5 个用户, 他们分属于这个服务供应商的 3 个不同的客户。用户与服务的交互是通过客户资料文档中为员工所定义的服务配置进行的。

多重租赁性是这样一种情况, 许多不同的用户访问相同的系统, 并且定义的系统体系结构允许系统资源的有效共享。不过, 对于每个用户来说必须感觉他们在唯一地使用系统资源。多重租赁性涉及将系统设计为在系统功能和系统数据之间有一个绝对的分

离。因此, 应该把系统设计成所有的操作都是无状态的, 以便共享这些操作。数据要么应该由客户端提供, 要么应该在存储系统或数据库中可以被所有的系统实例访问。

多重租赁系统的一个特殊问题是数据管理。提供数据管理最简单的办法是让每一个客户

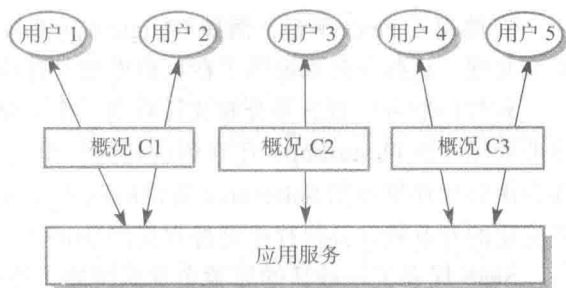


图 17-16 软件系统作为服务所提供的配置

有自己的数据库，这个数据库可以按照他们的想法使用和配置。不过，这需要服务提供商维护许多不同的数据库实例（每个客户一个），并使这些数据库在需要的时候是可用的。

另外一种方法是，服务提供商可以使用一个单一的数据库，在数据库中虚拟地隔离不同的用户。图 17-17 说明了这一点，可以看到数据库入口还有一个“租客标识符”，这个标识符将数据入口关联到特定的用户上。通过使用数据库视图，你可以为每一个服务的客户提取出数据入口，以此提供给此客户单位中的用户一个虚拟的个人数据库。使用上面提到的配置特征，可以扩展这种方法来满足具体的客户需要。

租 户	关键字	名 字	住 址
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J.Bowie	56, Mill St, Starville
592	PP37	R.Burns	Alloway, Ayrshire

图 17-17 多重租赁数据库

可伸缩性是系统可以处理不断增加的用户数量而不降低向每个用户提供的整体的 QoS（服务质量）的一种能力。通常，当在 SaaS 的环境下考虑可伸缩型时，我们采用的是“增加扩展”而不是“增强扩展”。回忆一下，“增加扩展”是指增加更多的服务器，由此也增加可以并行处理的事务数量。可伸缩性是个复杂的话题，这里不展开详细讨论，但是可伸缩软件的一般准则如下。

1. 所开发的应用要使得每一个构件都实现为简单的无状态服务，每一个服务都可以运行在任何一个服务器上。因而在单一事务过程中，用户可以与运行在多个不同服务器上的相同服务的多个实例交互。

2. 使用异步交互来设计系统，这样应用程序就不需要等待交互的结果（比如一个读请求）。这将允许应用程序在等待交互结束的同时可以继续执行当前的工作。

3. 管理资源，比如网络和数据库连接，将所有资源作为一个公用池，因此不让任何一个服务器游离在资源之外。

4. 设计你的数据库以允许细粒度的加锁。即当只有记录的一部分在使用时没必要锁定数据库中的所有记录。

5. 使用云 PaaS 平台，例如 Google App Engine（Sanderson 2012）或其他 PaaS 平台进行系统实现。这些平台都提供了在负载增加时自动扩展系统的机制。

软件即服务的概念是分布式计算的一个主要模式转变。我们已经看到了很多消费软件和专业应用（如 Photoshop）迁移到这种交付模式。越来越多的企业正在将他们自己的系统替换为由外部提供商如 Salesforce 提供的基于云的 SaaS 系统，例如 CRM 和库存系统。开发业务应用的专业软件公司程序更喜欢提供 SaaS，因为它简化了软件更新和管理。

SaaS 代表了一种新的思考企业系统的工程化的方式。以面向用户服务的方式来思考系统总是有用的，但是直到 SaaS 的出现才变成现实，这其中包含在实现系统时使用不同的抽象（例如对象）。如果用户和系统抽象之间能够更加匹配，那么所得到的系统就可以更容易理解、维护和演化。

要点

- 分布式系统的好处在于它可以调整以应对日益增长的需求，可以持续地向用户提供

服务（即使系统的某些部分失效），而且使资源得到共享。

- 分布式系统的设计需要考虑的问题包括透明性、开放性、可伸缩性、信息安全、服务质量和失效管理。
- 客户-服务器系统是分布式系统，其中系统构建为层次结构，表示层在客户端上实现。服务器提供数据管理、应用处理和数据库服务。
- 客户-服务器系统可能有许多层，系统的不同层分布在不同的计算机上。
- 分布式系统的体系结构模式包括主从体系结构、两层和多层客户-服务器体系结构、分布式构件体系结构和对等体系结构。
- 分布式构件体系结构需要中间件来处理构件间的通信并允许在系统中增加和删除构件。
- 对等（P2P）体系结构是分散式体系结构，其中没有区分客户端和服务端。计算可以分布在不同机构的很多系统中。
- SaaS 是把应用部署为瘦客户-服务器系统的一种方法，其中客户端是 Web 浏览器。

阅读推荐

《Peer-to-Peer: Harnessing the Power of Disruptive Technologies》这本书尽管没有太多关于 P2P 体系结构的内容，但是它是对 P2P 计算的一个很棒的介绍，也讨论了在多个 P2P 系统中所采用的结构和方法。（A. Oram (ed), O'Reilly and Associates Inc., 2001）

《Turing software into a service》是一篇综述文章，讨论了面向服务计算的一些准则。不像很多此主题的其他文章，它揭示了隐藏于很多标准背后的准则。（M. Turner, D. Budgen and P. Brereton, IEEE Computer, 36 (10), October 2003）dx.doi.org/10.1109/MC.2003.1236470

《Distributed Systems, 5nd edition》是一本讨论分布式系统设计和实现的综合性教科书。它涵盖了 P2P 系统和移动系统。（G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Addison-Wesley, 2011）

《Engineering Software as a Service: An Agile Approach Using Cloud Computing》这本书和作者的在线课程的主题相配套，是一本很好的针对初学者的实用性书籍。（A. Fox and D. Patterson, Strawberry Canyon LLC, 2014）<http://www.saasbook.info>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap17/>

支持视频的链接: <http://software-engineering-book.com/videos/requirements-and-design>

练习

- 17.1 如何理解可伸缩性？讨论“增强伸缩”和“增加伸缩”的区别，解释何时使用这些不同的可伸缩性方法。
- 17.2 为什么分布式软件系统要比所有的系统功能都实现在单一计算机上的集中式软件系统更复杂？
- 17.3 用一个远程过程调用的例子来说明中间件是如何在分布式系统中协调各计算机之间的交互的。
- 17.4 在客户-服务器系统体系结构中，胖客户机和瘦客户机方法的根本差别在哪里？

- 17.5 假如要求你设计一个需要强身份认证和权限认证的信息安全系统。该系统必须保证系统中各个部分之间的通信不能被攻击者拦截和读取。提出该系统最合适的客户-服务器体系结构,并解释为什么你如此设计,在客户端和服务器之间功能应该如何分布?
- 17.6 假设要开发一个股票信息系统,向客户提供对公司信息的访问,并能够利用仿真系统对各种投资情形做出评估。不同的客户会根据他们的经验而采取不同的投资方式,而且购买的股票类型也是不同的。为该系统提出一个客户-服务器体系结构,指出各个功能在哪里实现,并对该模型做出一些判断。
- 17.7 使用分布式构件的方法,为国家剧院订票系统设计一个体系结构。用户可以查询空座以及预订一组剧场的座位。系统应当支持退票,因此,人们可以退掉他们的票以便剧院在最后一分钟将票转售给其他客户。
- 17.8 分别给出分散的和半集中的对等体系结构的两个优点和两个缺点。
- 17.9 为什么以将软件部署为一种服务可以减少公司的IT支持成本?如果使用此部署模型,可能会产生什么额外成本?
- 17.10 你的公司想要从使用桌面应用程序转变为以服务的方式访问远程相同的功能。确定3个可能出现的风险,并建议如何减少这些风险。

参考文献

- Bernstein, P. A. 1996. "Middleware: A Model for Distributed System Services." *Comm. ACM* 39 (2): 86-97. doi:10.1145/230798.230809.
- Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. *Distributed Systems: Concepts and Design*, 5th ed. Harlow, UK: Addison-Wesley.
- Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates.
- McDougall, P. 2000. "The Power of Peer-to-Peer." *Information Week* (August 28, 2000). <http://www.informationweek.com/801/peer.htm>
- Oram, A. 2001. "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology." Sebastopol, CA: O'Reilly & Associates.
- Orfali, R., D. Harkey, and J. Edwards. 1997. *Instant CORBA*. Chichester, UK: John Wiley & Sons.
- Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.
- Sanderson, D. 2012. *Programming with Google App Engine*. Sebastopol, CA: O'Reilly Media Inc.
- Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.
- Tanenbaum, A. S., and M. Van Steen. 2007. *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall.
- Wallach, D. S. 2003. "A Survey of Peer-to-Peer Security Issues." In *Software Security: Theories and Systems*, edited by M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, 42-57. Heidelberg: Springer-Verlag. doi:10.1007/3-540-36532-X_4.

面向服务的软件工程

目标

本章的目标是介绍面向服务的软件工程，利用该方法可构建使用 Web 服务的分布式应用。阅读完本章后，你将：

- 理解关于 Web 服务、Web 服务标准以及面向服务体系结构的基本概念；
- 理解 RESTful 服务的思想以及 RESTful 服务与基于 SOAP 的服务的重要区别；
- 理解以产生可复用的 Web 服务为目的的服务工程过程；
- 理解如何使用基于工作流的服务组合来创建支持业务过程的面向服务的软件。

在 20 世纪 90 年代，网络技术的发展彻底改变了机构的信息交流方式。客户计算机可以访问它们机构外的远程服务器来获得信息。但是，这种访问完全是通过 Web 浏览器进行的，要想使用其他程序来对信息库进行直接访问是不可行的。也就是说，在服务器之间进行任意连接（比如一个程序从不同的提供商查询多个目录）是不可能做到的。

为了解决这个问题，人们提出了 Web 服务的概念，即允许程序访问和更新位于网络上的资源。使用 Web 服务，机构通过定义和建立一个 Web 服务接口就可让自己的信息被别的程序访问。这个接口定义可用的数据和如何访问和使用这些数据。

更一般的情况，Web 服务是一个标准的计算资源或信息资源的表示，这些资源可以被其他程序使用。这些可能是信息资源，例如一个零件目录；也可以是计算机资源，例如一个专门的处理器；或者是存储资源，例如，存档服务能够实现对商店数据的长期、可靠存储，依据法律这些组织数据必须保持多年。

Web 服务是更一般化的服务概念的一个实例，对于一般的服务概念，由 Lovelock 等 (Lovelock et al.1996) 给出的定义是这样的：

由一方向另外一方提供的行动或能力。尽管这个过程可能是与一个有形的产品联系在一起的，但是能力本质上是无形的，一般不会产生对任何生产要素的拥有权^①。

服务是自然开发的软件构件，构件模型在本质上是一些和 Web 服务相关的标准。因此，一个 Web 服务可以被定义为：

一个松耦合、可复用的软件构件，封装了离散的功能，该功能是分布式的并且可以被程序访问。Web 服务是通过标准互联网和基于 XML 的协议被访问的服务。

如在基于构件的软件工程 (Component-Based Software Engineering, CBSE) 中所定义的，服务和软件构件之间的一个重要的区别是，服务应该总是独立的和松耦合的，即它们应该总是以相同的方式运行，而与它们的执行环境无关。它们不应该依赖于一些在功能性和非功能性行为上不一致的外部构件。因此，Web 服务没有“请求”接口，在 CBSE 中，“请求”接口定义必须出现的其他系统构件。Web 服务接口是定义服务功能和参数的简单的“提供”接口。

^① Lovelock,C.,Vandermerwe,S.and Lewis,B.(1996).Services Marketing.Englewood Cliffs,NJ: Prentice Hall.

面向服务的系统是一种开发分布式系统的方法，系统构件是独立的服务，执行在位于不同地理位置的计算机上。服务是无关于平台和实现语言的。可以通过组合本地服务和不同提供商提供的外部服务来构建软件系统，系统中的服务间可以无缝交互。

正如第 17 章中讨论的，“软件即服务”和“面向服务的系统”并不一样。“软件即服务”意味着通过网络远程提供软件功能给用户，而不是通过安装在用户计算机上的应用。“面向服务的系统”是使用可复用的服务构件来实现的，这些构件可以被其他程序访问，而不是直接被用户访问。作为服务的软件可以通过使用面向服务的系统来实现。然而，并没有必要通过这种方式来实现软件以提供用户服务。

采用面向服务的方法对软件工程有如下这些重要的好处。

1. 组织内部或者外部的服务提供商都可以提供服务。假如这些服务符合某些特定的标准，组织能够通过集成一系列服务提供商提供的服务来创建应用。例如，一个制造业公司可以直接连接到他的供应商所提供的服务。

2. 服务提供商会公开关于服务的信息，所以任何获得授权的用户都可以使用相应服务。服务的提供商和服务的用户并不需要在服务被合并到某个应用程序之前协商服务能做什么。

3. 应用能够延迟服务绑定直到这些服务被部署或者执行。因此，原则上一个使用股票价格服务的应用能够在系统执行的过程中动态地改变服务提供商。这意味着应用可以是反应式的，并且根据他们的执行环境的变化来调整操作。

4. 任意创建新的服务是可行的。通过一种创新的方式连接已有的服务，服务提供商可以挖掘出新的服务。

5. 服务的用户能够根据使用而不是提供商提供的服务来付费。因此，应用的作者能够在需要使用外部服务的时候付费，而不是购买一个很少使用但是昂贵的构件。

6. 应用可以变得更小，这对于移动设备来说非常重要，因为移动设备的处理能力和内存都有限。一些计算密集型的处理和异常处理可以被迁移到外部的服务上。

面向服务的体系结构是松耦合的体系结构，服务绑定在执行阶段是可以改变的。这意味着可能会在不同的时间执行着服务的不同的但等价的版本。有些系统完全是使用 Web 服务来构建的，而其他一些系统是结合了 Web 服务和本地开发的构件。为了说明混合使用服务和构件的应用是如何组织的，考虑以下情形。

车载信息系统为驾驶人员提供关于天气、交通状况、本地信息等内容。这是链接到车上的无线电装置上的，这样信息作为信号可以在专设的频道上传输。轿车上配备 GPS 接收装置来发现自己所在的位置，基于这个位置信息，系统访问一系列信息服务。信息可以使用驾驶员指定的语言来传输。

图 18-1 说明了这样一个系统的可能组成。车载软件系统包括 5 个模块。它们处理与驾驶人员、与报告车辆位置的 GPS 接收装置，以及与车上的无线电接收装置的通信。Transmitter（传递器）和 Receiver（接收器）两模块处理所有与外部设备之间的通信。

车辆与外部移动信息服务通信，这个外部信息服务又聚集了很多来自其他模块的一些服务，这些服务提供了诸如气象、交通信息、本地设施等信息。位于不同地点的不同提供商提供这些服务，车载系统用外部的发现服务来定位本地最合适的信息服务并绑定它。移动信息服务也使用发现服务来绑定合适的气象、交通以及设施等服务。所收集的信息被发回汽车后，通过一个服务被翻译成驾驶人员喜欢的语言。

这个例子很好地说明了面向服务方法的一个重要优势——在系统编程阶段和部署阶段无

须决定使用哪个服务提供商,也无须知道要访问什么特殊的服务。当汽车随处行驶的时候,车载软件使用发现服务来寻找最为合适的信息服务并绑定它。由于使用了翻译服务,它能够跨越地域而让不懂当地语言的驾驶人员了解到当地的信息。

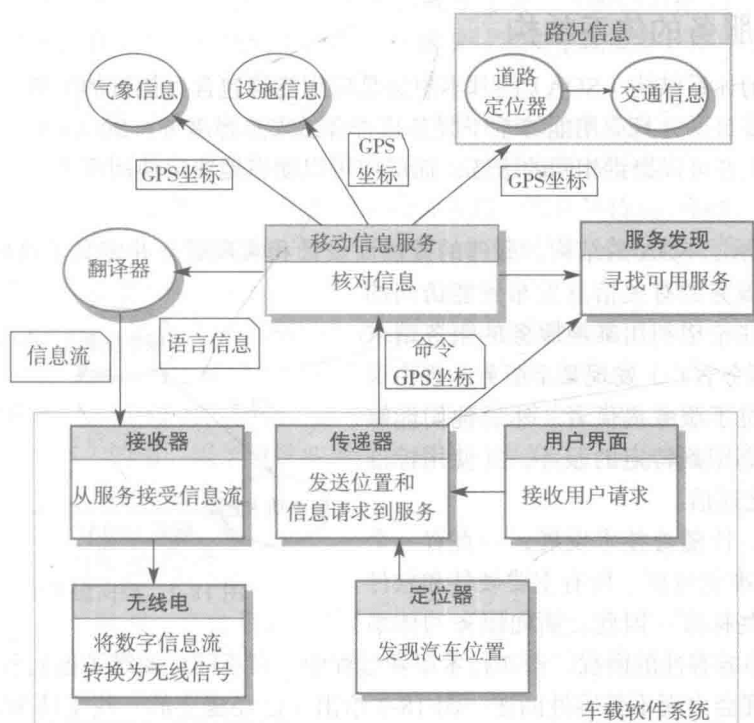


图 18-1 一个基于服务的车载信息系统

软件工程的面向服务方法是一种新的软件工程范式,在作者看来,是与面向对象软件工程同等重要的软件工程的一个重要进展。这种范式的转变将加速云计算和移动计算系统的发展,这已经并将继续对系统产品和业务流程产生深远影响。Newcomer 和 Lomow(Newcomer and Lomow 2005)在他们关于 SOA 的书中总结了面向服务的方法的潜力:

由于关键技术的集成和 Web 服务普遍采用的驱动,面向服务的企业承诺极大地提高企业的灵活性,加快新产品和服务推向市场的步伐,降低 IT 的成本并改善运作的效率^①。

基于服务的应用开发使得公司和其他组织机构能够相互协作以及利用其他机构的业务功能。因此,那些包含跨公司的大量信息交换的系统能够轻易实现自动化,例如供应链系统中,一家公司预订了另一家公司的商品。正如在 18.4 节中讨论的,通过使用标准编程语言或者专业的工作流语言,连接来自诸多提供商的服务,能够创建基于服务的应用。

服务提供和实现方面初始的工作很大程度上受到了软件产业在构件标准方面失败的影响。因此,这件事是标准驱动的,所有主要的产业界企业都参与到了标准的开发之中。这一做法推动了一整套标准(WS*standards)和面向服务的体系结构(Service-Oriented Architecture, SOA)的概念。为了创造基于服务的系统,这些标准被作为体系结构而提出,同时所有的服务通信都是基于标准的。然而,提出的这些标准复杂并且有显著的运行开销。

① Newcomer,E.and Lomow,G.(2005).Understanding SOA with Web Services.Boston:Addison-Wesley.

这个问题导致许多公司采用了一个基于 RESTful 服务的替代的体系结构。相比于面向服务的体系结构，RESTful 方法更加简单，但是相对不适合提供复杂功能的服务。这两种体系结构在本章中都有讨论。

18.1 面向服务的体系结构

面向服务的体系结构（SOA）的基本想法是应用能够包含可执行的服务。服务有定义明确的、公开的接口，并且应用能够基于服务是否合适来选择服务。SOA 的一个重要思想是，不同的服务提供者可以提供相同的服务，而应用可以动态地决定使用哪个服务提供者提供的服务。

图 18-2 展示了 SOA 的结构。服务的提供商设计和实现服务并定义了这些服务的接口。他们也将这些服务的有关信息发布到能访问的注册表上。那些希望利用某项服务的服务请求者（有时叫作服务客户）发现某个服务的规格说明，从而也定位了服务提供者。然后他们能够将自己的应用绑定到特定的服务，并使用标准的服务协议与之通信。

从一开始，伴随着技术发展，一直有一个活跃的 SOA 标准化过程。所有主要硬件和软件公司都接受这些标准。因此，面向服务的体系结构没有受到不兼容性的困扰，而在技术革新过程中，在不同厂商维护他们各自的技术版权的地方，通常都会出现不兼容性问题。图 18-3 给出了已经建立的一些支持 Web 服务的关键标准。

Web 服务协议覆盖了面向服务的体系结构的所有方面：从基本的服务信息交换（SOAP）机制到编程语言标准（WS-BPEL）。这些标准全部基于可扩展标记语言（eXtensible Markup Language, XML）——一种人和计算机都可识别的标记语言，它允许定义结构化的数据，其中文本用一个有意义的标识符来标记。XML 有一系列的支持技术，例如用于模式定义的 XSD，它用于扩展和处理 XML 描述。Erl（Erl 2004）概要描述了 XML 技术以及它们在 Web 服务中作用。

简要地说，面向 Web 服务的体系结构的主要标准有以下这些。

1. SOAP。这是一个支持服务之间通信的消息交换标准。它定义服务之间消息传递的必需的和可选的构件。SOA 中的服务有时被叫作基于 SOAP 的服务。
2. WSDL。Web 服务定义语言（Web Service Description Language, WSDL）制定服务接口的标准。它给出了服务是如何操作的（操作名、参数、它们的类型）以及必须定义的服务绑定。
3. WS-BPEL。这是一个工作流语言的标准，工作流语言用来定义包括多个不同服务的

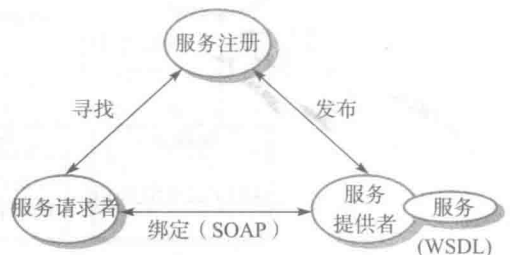


图 18-2 面向服务的体系结构

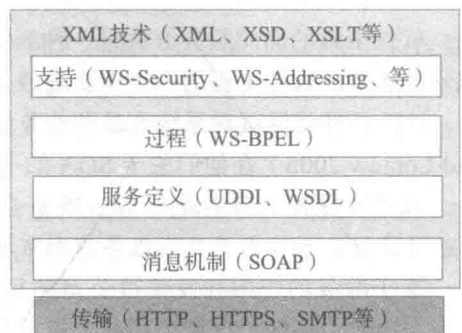


图 18-3 Web 服务标准

过程程序。18.3 节讨论过程程序的概念。

通用描述、发现和集成 (Universal Description, Discovery, and Integration, UDDI) 标准定义了服务规格说明的组成成分, 其目的是帮助潜在的用户来发现服务的存在。该标准的目的是允许企业建立带有 UDDI 描述的注册表的服务注册。许多公司在 21 世纪初期就建立了 UDDI 注册服务, 但是用户更倾向于使用标准搜索引擎来发现服务。所有公开的 UDDI 注册现已全部被关闭。

主要的 SOA 标准受到一系列 SOA 的更专业方面的标准的支持。存在非常多的支持标准, 因为它们希望在不同类型的应用中支持 SOA。这些标准的例子包括:

1. WS-Reliable Messaging, 一个确保消息将会传递一次且只传递一次的消息交换标准。
2. WS-Security, 是一套支持 Web 服务信息安全性的标准, 包括指定信息安全政策定义的标准和覆盖数字签名使用的标准。
3. WS-Addressing, 定义在一个 SOAP 消息中如何表达地址信息。
4. WS-Transactions, 指定在分布式服务之间的事务应该怎样协调。

Web 服务标准是一个很大的话题, 这里无法详细地讨论它们。有兴趣的读者可以阅读 Erl (Erl 2004, 2005) 的书, 以了解这些标准的概况。有关它们的详细描述也可以通过 Web 上的公开文档 (W3C 2013) 获得。

18.1.1 SOA 中的服务构件

正如 17.1 节中解释的, 在分布式计算系统中, 消息交换是一个用于协同行为的重要机制。SOA 中的服务通过交换 XML 格式的消息来通信, 这些消息使用一些标准互联网传输协议 (如, HTTP、TCP/IP) 来传递。

服务定义它对另一个服务的需要, 方法是在消息中陈述需求并发送给相应的服务。接收方解析消息, 执行计算, 完成后发送一个回复, 作为消息, 给请求的服务。提出请求的服务于是解析回复的消息, 提取所需要的信息。与软件构件不同, 服务不使用远程过程调用或远程方法调用来访问与其他服务关联的功能。

当你想要使用 Web 服务的时候, 你需要知道服务在哪里 (它的 URI) 以及接口细节。这些会在服务描述中说明, 服务描述用 XML 语言书写, 被称为 Web 服务描述语言 (WSDL)。WSDL 规格说明定义了有关 Web 服务的 3 个方面: 服务做什么, 如何通信, 在哪里能找到它。

1. WSDL 文档的 “what” 部分, 称为接口, 指定服务所支持的操作, 并且定义服务发送和接收的消息的格式。

2. WSDL 文档的 “how” 部分, 称为一个绑定, 把抽象接口映射到一组具体的协议上。绑定指定了如何与一个 Web 服务通信的技术细节。

3. WSDL 文档的 “where” 部分, 描述一个特定的 Web 服务实现的位置 (它的端点)。

WSDL 概念模型 (见图 18-4), 给出了服务描述的元素。其中每个元素用 XML 表达并且可以在单独的文件中完成。这些元素是:

1. 一个介绍性的部分, 通常定义 XML 所使用的名字空间, 可能还会包含一个提供有关服务的额外信息的文档段。

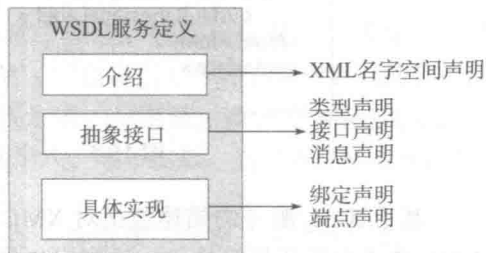


图 18-4 WSDL 规格说明的组织结构

2. 一个可选的对于服务所交换的消息中所使用的类型的描述。
3. 一个对于服务接口（即服务为其他的服务或用户提供的操作）的描述。
4. 一个对于服务所处理的输入和输出消息的描述。
5. 一个对于服务所使用的绑定（即将用于发送和接收消息的消息通信协议）的描述。默认是 SOAP，但也可以指定其他绑定。绑定说明如何将与服务关联的输入和输出消息封装在消息中，并且指定所使用的通信协议。绑定也可以指定支持信息，诸如安全证书或事务标识符将包括在内。
6. 一个端点规格说明，这是服务的物理位置，表示为统一资源标识符（URI），即可以在互联网上访问的资源地址。

图 18-5 展示了一个简单服务的部分接口，给定日期和地点（精确到某个国家的某个城镇），该服务可以返回当地在指定日期的最高和最低温度。输入的消息也确定了是按照摄氏度还是华氏度来返回温度。

定义一些所使用的类型。假设命名空间前缀“ws”指向XML schemas的命名空间URI，与这个定义相关联的命名空间前缀是weathns。

```

<types>
  <xs:schema targetNamespace = "http://. ./.weathns"
    xmlns:weathns = "http://. ./.weathns" >
    <xs:element name = "PlaceAndDate" type = "pdrec" />
    <xs:element name = "MaxMinTemp" type = "mmtrec" />
    <xs:element name = "InDataFault" type = "errmess" />

    <xs:complexType name = "pdrec"
      <xs:sequence>
        <xs:element name = "town" type = "xs:string" />
        <xs:element name = "country" type = "xs:string" />
        <xs:element name = "day" type = "xs:date" />
      </xs:sequence>
    </xs:complexType>

    这里是MaxMinType和InDataFault的定义。

  </schema>
</types>

```

现在定义接口及其操作。这里，只有一个返回最高和最低温度的操作。

```

<interface name = "weatherInfo" >
  <operation name = "getMaxMinTemps" pattern = "wsdl:in-out" >
    <input messageLabel = "In" element = "weathns:PlaceAndDate" />
    <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
    <outfault messageLabel = "Out" element = "weathns:InDataFault" />
  </operation>
</interface>

```

图 18-5 一个 Web 服务的部分 WSDL 描述

基于 XML 服务的描述包括对 XML 命名空间的定义。命名空间中的标识符优先级高于 XML 描述中的任何标识符，这样可以区分 XML 描述中位于不同部分但有相同命名的标识符。为了理解本书中的例子，你并不需要了解命名空间的细节。你只需要知道命名前会加上命名空间标识符，也就是说 namespace:name 应该是唯一的。

在图 18-5 中, 描述的第一部分展示了用于服务规格说明的部分元素和类型定义, 定义了元素 PlaceAndDate、MaxMinTemp 和 InDataFaul。在这里给出了 PlaceAndDate 的规格说明, 读者可以认为其记录了 3 个值域, 分别为 town、country 和 date。类似地可以定义 MaxMinTemp 和 InDataFaul。

描述的第二部分展示了服务接口是如何定义的。虽然服务中可以包含的操作数量并没有限制, 但是在这个例子中, 服务 weatherInfo 只有一个操作。服务 weatherInfo 的操作符合 in-out 模式, 该模式每接收一个消息就发出一个消息。WSDL 2.0 规格说明允许一系列消息交换模式, 例如 in-only、in-out、out-only、in-optional-out、out-in。输入和输出的消息 (涉及先前类型部分的定义) 就定义好了。

WSDL 中服务接口的定义是: 关于服务签名 (即服务的操作和操作的参数) 的描述。描述并不涉及服务的语义以及服务的非功能特性, 比如性能和可依赖性。如果想要使用服务, 就必须弄明白服务到底能做什么以及输入、输出消息的含义, 而服务的性能和可依赖性需要通过实验来找出。服务实际做了什么有可能被误解, 而有意义的命名和文档可以帮助理解服务的功能。

用 XML 书写的完整服务描述, 读起来是冗长、琐碎且枯燥的。WSDL 描述现在很少手工书写, 描述中的大多数信息都是自动生成的。

18.2 RESTful 服务

Web 服务和面向服务的软件工程的初始研发都是基于标准的, 并且服务间交换的消息是基于 XML 的。这是开发复杂服务、动态服务绑定、控制服务质量以及服务可依赖性的一种一般性方法。然而, 随着服务的开发, 人们慢慢发现绝大多数服务都是单一功能的, 这些服务有着相对简单的输入和输出接口。服务使用者对动态绑定和使用多个服务提供商并不真正感兴趣。他们几乎不使用关于服务质量、可靠性等 Web 服务标准。

问题是, Web 服务标准是“重量级”的标准, 这些标准有时候过于一般化和低效。实现这些标准需要处理大量的 XML 消息, 包括产生、传输和解析 XML 消息。这降低了服务间通信速率, 对于高吞吐率的系统来说, 需要额外的硬件来实现要求的服务质量。

为了应对这种情况, 人们开发了 Web 服务体系结构, 这是作为替代的一种“轻量级”的方法。这种方法符合 REST 体系结构的风格, 其中 REST 意为表示层状态转化 (Representational State Transfer) (Fielding 2000)。REST 是一种基于从服务器向客户端传输可识别资源的体系结构风格。这种风格作为一个在整体位于网络之下, 并且是一个比 SOAP/WSDL 更简单的实现 Web 服务接口的方法。

资源是 RESTful 体系结构中的基本元素。从本质上来说, 资源仅仅是一种数据元素, 例如目录、医疗记录或者一份文件 (如本章节)。总之, 资源可以有多种表现形式; 换言之, 它们能够以不同的格式存在。例如, 本章有 3 种表现形式, 分别是: 微软的 Word 的形式, 用于编辑; PDF 的形式, 用于网页展示; InDesign 的形式, 用于出版。这些表现形式底层的由文本和图片构成的逻辑资源是一致的。

在 RESTful 体系结构中, 任何事物都可以表示成一种资源。资源有一个唯一标识符, 也就是它们的 URL。资源有一点像对象, 有 4 种基本的多态操作, 如图 18-6a 中所示。

1. 创建——创建资源。
2. 读取——返回资源的一种表现形式。

3. 更新——改变资源的值。

4. 删除——使资源不可访问。

例如，网络是一个符合 RESTful 体系结构的系统。网页是资源，网页的 URL 是它的统一标识符。

网络协议 http 和 https 都是基于 POST、GET、PUT 和 DELETE 这 4 种动作。这 4 种动作映射到 4 种基本资源操作，如图 18-6 b 所示。

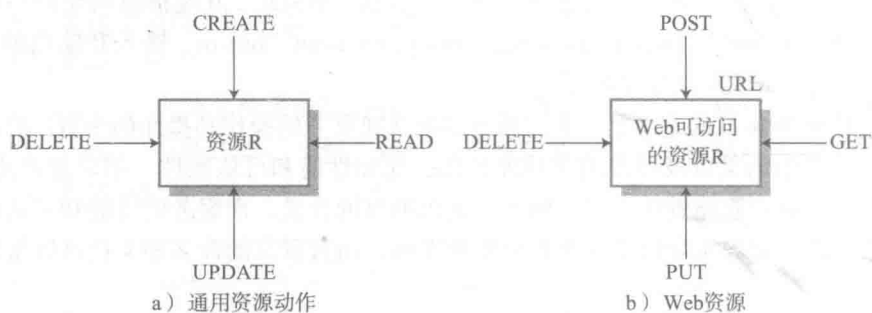


图 18-6 资源和动作

1. POST 用于创建资源。它有创建资源的相关数据。

2. GET 用于读取资源的值，并以特定的表现形式返回给请求者，比如 XHTML，可以在网络浏览器中呈现。

3. PUT 用于更新资源的值。

4. DELETE 用于删除资源。

在某种程度上，所有服务都在操作数据。例如，18.2 节中描述的服务使用了天气信息数据库，它会根据给定的数据返回某地最高和最低温度。基于 SOAP 的服务对这个数据库执行操作，从而返回特定的值。RESTful 服务 (Richardson and Ruby 2007) 直接访问数据。

当使用 RESTful 方法时，都是利用数据的 URL 来暴露和访问数据的。RESTful 服务使用 http 或者 https 协议，只有 POST、GET、PUT 和 DELETE 这 4 种动作。因此，数据库中各地的天气数据可以通过 URL 来访问，这些 URL 如下所示：

`http://weather-info-example.net/temperatures/boston`

`http://weather-info-example.net/temperatures/edinburgh`

这些 URL 会调用 GET 操作并返回一个关于最高和最低温度的列表。为了请求一个特定日期的温度，可以使用如下的 URL 查询：

`http://weather-info-example.net/temperatures/edinburgh?date=20140226`

鉴于世界上可能会有几个地方使用相同的名字，URL 查询也可以用来区分请求：

`http://weather-info-example.net/temperatures/boston?date=20140226&country=USA&state="Mass"`

RESTful 服务和基于 SOAP 的服务之间的一个重要区别是，RESTful 服务是不完全基于 XML 的。所以，当请求、创建或者改变一个资源时，就应该确定其表现形式。这一点对 RESTful 服务来说非常重要，因为 RESTful 服务可能会使用 JSON (Javascript Object Notation) 和 XML。以 JSON 表示的资源比基于 XML 表示的资源处理起来更高效，从而可以降低服务调用中的开销。因此，上文关于波士顿最高和最低温度的请求可能会返回如下

信息:

```
{
  "place": "Boston",
  "country": "USA",
  "state": "Mass",
  "date": "26 Feb 2014",
  "units": "Fahrenheit",
  "max temp": 41,
  "min temp": 29
}
```

在 RESTful 服务中, 一个 GET 请求的响应可能包含多个 URL。因此, 如果一个请求的响应是一系列资源, 那么响应中可以包含这些服务的 URL。发起请求的服务可能会以自己的方式处理请求。因此, 如果请求天气信息时, 所请求的地点的名字有重复, 可能会返回所有匹配该地点的 URL。例如:

`http://weather-info-example.net/temperatures/edinburgh-scotland`

`http://weather-info-example.net/temperatures/edinburgh-australia`

`http://weather-info-example.net/temperatures/edinburgh-maryland`

RESTful 服务的一个基本设计原则是服务应该是无状态的。换言之, 在一个交互会话中, 资源本身不应该包含任何状态信息, 比如最后一次请求的时间。反而, 所有必要的状态信息都应该返回到请求方。如果在后续的请求中需要状态信息, 那么这些信息应该由请求方返回到服务器。

在过去几年中由于移动设备的广泛使用, RESTful 服务已经被更广泛地使用了。移动设备的处理能力有限, 所以 RESTful 服务的开销越小, 系统性能越好。为一个已有的网站实现一个 RESTful 的 API 通常是相对简单的, 所以 RESTful 服务对已有的网站来说也很容易使用。

然而, RESTful 方法也存在一些问题:

1. 当一个服务有着复杂的接口并且不是一个简单的资源, 那么设计一系列 RESTful 服务来代表该服务的接口将变得困难。

2. 并没有关于 RESTful 接口描述的标准, 因此服务使用者必须依赖于非正式的文档来理解接口。

3. 当使用 RESTful 服务时, 必须实现你自己的基础设施, 以便监视和管理服务质量和可靠性的基础设施。基于 SOAP 的服务有额外的基础设施支持标准, 例如 WS-Reliability 和 WS-Transactions。

Pautasso 等 (Pautasso, Zimmermann, and Leymann 2008) 讨论了什么时候使用 RESTful 和基于 SOAP 的方法。然而, 通常可以提供基于 SOAP 和 RESTful 的接口给相同的服务或资源 (见图 18-7)。这种双重方法现在在云服务提供商中比较常见, 如微软、谷歌和亚马逊。服务的使用者就可以选择最适合自己的服务访问方法。

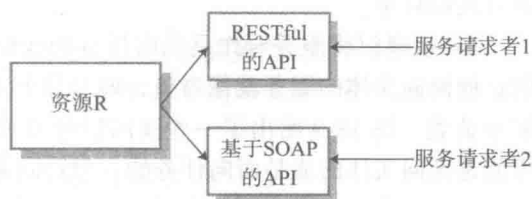


图 18-7 RESTful 和基于 SOAP 的 API

18.3 服务工程

服务工程是开发服务的过程，这种服务在面向服务的应用中是可复用的。它和构件工程非常类似。服务工程师必须确保服务代表可复用的抽象，能用于不同系统的抽象。他们要设计开发与此抽象关联的有用的一些功能，而且确保服务是健壮的和可靠的。他们必须为服务提供文档，以便需要的用户能够发现和了解该服务。

在服务工程过程中有如下 3 个逻辑阶段（如图 18-8 所示）。

1. 可选服务识别，在此阶段识别那些需要实现的服务，并定义服务需求。

2. 服务设计，在此阶段设计逻辑和 WSDL 的服务接口（基于 SOAP 和 / 或 RESTful 的）。

3. 服务实现和部署，在此阶段实现并测试服务，使之可用。

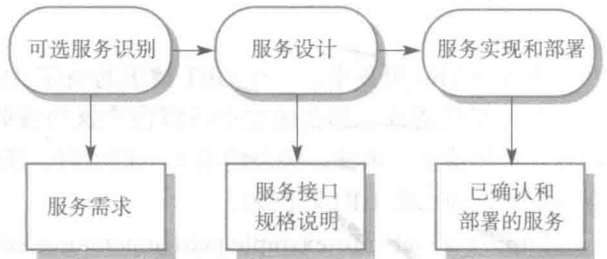


图 18-8 服务工程过程

如第 16 章中提到的，可复用构件的开发可能是在已实现的并在应用程序中使用的现有构件的基础上开始的。服务同样如此，此过程的起点往往是一个现有的服务或者要转化为服务的构件。在这种情况下，设计过程涉及归纳现有构件，删除应用程序特定的功能。实现意味着通过添加服务接口改写原来的构件，并实现所需要的泛化。

18.3.1 可选服务识别

面向服务计算的基本理念是服务应该支持业务过程。由于每个机构都有很多的过程，因此存在许多可能的服务可以加以实现。可选服务识别需要理解和分析组织机构的业务过程，来决定哪些可复用服务可能需要实现以支持这些过程。

Erl (Erl 2005) 建议定义如下 3 种基本服务类型。

1. 实用服务。这些服务实现某些一般性的功能，可用于不同的业务过程。一个实用服务的例子是货币转换服务，通过访问它可以计算一种货币（例如美元）对另外一种货币（例如欧元）的兑换。

2. 业务服务。这些服务是与特殊业务功能相关的服务。大学里的业务功能的例子是学生为一门课程注册登记。

3. 协同或过程服务。这些服务是为支持更一般的业务过程，这些业务过程总是包含不同的角色和活动。公司里的协同服务的例子是订货服务，允许完成一个包含厂商、产品以及付款方式的订单。

Erl 也建议将服务看作是面向任务的或面向实体的。面向任务的服务是与某项活动关联的；而面向实体的服务就像对象，即与某个业务实体关联，这样的业务实体的例子是一张求职申请表。图 18-9 给出了一些面向任务的和面向实体的服务的例子。实用服务或业务服务可能是面向实体的或是面向任务的，但协同服务总是面向任务的。

可选服务识别阶段的目标应该是找出那些逻辑上相关的、独立的且可复用的服务。Erl 的分类在这方面是有帮助的。它给出了如何通过对业务实体和业务活动的观察来发现可复用的服务。然而，识别可选服务有时是很困难的，因为你不得不想象这项服务将是如何使用

的。你必须考虑所有可能的候选项，然后通过一系列关于它们的问题来分析是否是有用的服务。能帮助你发现可复用的服务的问题包括以下这些。

1. 对于一个面向实体的服务，它是与单个用于不同业务过程的逻辑实体关联的吗？通常情况下在必须支持的实体上都执行哪些操作？这些操作是否和 RESTful 服务的 PUT、CREATE、POST 和 DELETE 操作相一致？

2. 对于一个面向任务的服务，该任务是在机构中由不同的人执行的吗？当提供单个支持服务时要发生不可避免的标准化问题，他们愿意接受吗？这些能够适应 RESTful 的模型吗？或者应该重新设计面向实体的服务吗？

3. 服务是独立的吗？也就是说，它在多大程度上依赖其他服务的可用性？

4. 服务必须维护状态吗？如果需要状态信息，那么这些信息必须存储在数据库中或者作为参数传递给服务。服务和所需的数据库之间将会有依赖关系，这将影响服务的复用性。总的来说，那些具有已传递状态的服务是易于复用的，因为不需要绑定数据库。

5. 服务能被机构外的客户使用吗？举例来说，一个与目录关联的面向实体的服务可以既由内部访问又可以由外部访问吗？

6. 服务的不同用户可能有不同的非功能性需求吗？如果有，那么就实现不止一个版本的服务。

	实用服务	业务服务	协同服务
任务	货币兑换器 员工定位器	确认申报表格 检查信用等级	过程费用申报 支付外部供应商
实体	文档翻译器 Web 表格到 XML 转换器	费用表格 学生申请表	

图 18-9 服务类别

这些问题的答案有助于我们去选择和精练那些将被实现为服务的抽象。然而，决定哪个是最好的服务并没有公式化的方法，因此可选服务识别是一个基于技术和经验的过程。

可用服务选择过程的输出是一组找到的服务以及相关的需求。功能性服务需求需要定义服务应该做什么。非功能性服务需求需要定义服务的信息安全性、性能和可用性需求。

为了帮助读者理解可选服务的识别和实现，考虑下面的例子：

一家卖计算机设备的大公司，已经为一些大客户的核准配置安排了特殊价格。为了方便自动订购，公司希望制作一个目录服务，允许客户选择他们需要的设备。与消费者目录不同，订单不是直接通过目录接口下达。相反，是通过每家公司的基于 Web 的采购系统下订单的，每家公司以 Web 服务来访问目录。绝大多数公司有自己的预算流程和订单批复流程，在下订单时必须遵循他们自己的流程。

目录服务是支持业务操作的面向实体服务的一个例子。目录服务的功能性需求如下。

1. 目录的特定版本将提供给每家用户公司。它包括由客户公司的员工所订购的配置和设备以及目录项的议定价格。

2. 目录应该允许客户公司职员下载目录版本以便脱机浏览。

3. 目录应该允许用户比较最多 6 个目录项的规格说明和价格。

4. 目录应该为用户提供浏览和搜索工具。

5. 目录的用户应该能够根据某一特定目录项的代码发现可预期的交付日期。

6. 目录的用户应该能够下“虚拟订单”，即所需的项可以为他们保留 48 小时。虚拟订单必须通过采购系统下达的一个真实订单来确定。确认消息必须在虚订单的 48 小时内收到。

除了这些功能性需求之外，目录还有许多的非功能性需求。

1. 访问目录服务的权限将限制为认可的机构职员。
2. 提供给一客户的价格和配置信息将是保密的，对任何其他客户来说是不可见的。
3. 目录在从格林尼治标准时间 0700 到格林尼治标准时间 1100 都将可用且不间断。
4. 目录服务在峰值负载时能够每秒处理高达 100 个请求。

注意这里没有有关目录服务响应时间的非功能性需求。这依赖于目录的大小和预计的并发用户的数量。因为这不是一个时间要求极高的服务，所以不需要在这个阶段指定它。

18.3.2 服务接口设计

一旦选择了可选服务，服务工程过程的下一个阶段就是设计服务接口。这包括定义与服务关联的操作以及它们的参数。如果使用基于 SOAP 的服务，就必须设计输入和输出消息。如果使用 RESTful 服务，就必须考虑需要的资源以及如何使用标准操作来实现服务操作。

服务接口设计的起点是抽象接口设计。确定与服务相关的实体和操作、服务的输入和输出，以及与这些操作相关的异常。然后需要思考抽象接口如何实现成基于 SOAP 或者 RESTful 的服务。

如果选择了基于 SOAP 的方法，就必须设计服务发出和收到的 XML 消息的结构。操作和消息是由 WSDL 编写的接口描述的基础。如果选择了 RESTful 方法，就必须设计如何将服务操作映射到 RESTful 操作上。

抽象接口设计从服务需求、定义操作名称和参数开始。在此阶段，也需要定义调用服务操作可能产生的异常。图 18-10 展示了实现了需求的目录操作。没有必要为这些列出细节，你会在设计过程的下一个阶段为其添加细节。

操 作	描 述
MakeCatalog	为特定客户创建一个目录的特殊版本。包括一个可选参数来创建目录的一个可下载的 PDF 版本
Lookup	显示与特殊目录项相关联的所有数据
Search	此操作接受逻辑表达式，根据此表达式搜索目录。显示能匹配表达式的所有项的一个列表
Compare	最多可提供 4 个目录项的 6 个特性以进行比较（例如，价格、尺寸、处理器速度等）
CheckDelivery	如果在某天预订，返回此项的预计的交付日期
MakeVirtualOrder	保存将由客户订购的项的数目，并为客户自己的采购系统提供各项的信息

图 18-10 目录服务操作

一旦建立了关于服务做什么的非正式描述，下一个阶段就会增加更多关于服务输入和输出的细节。在图 18-11 中，基于目录服务展示了这点，扩展了图 18-10 中的功能性描述。

定义异常以及这些异常如何传达给服务使用者是特别重要的。服务工程师并不知道他们的服务将如何被使用。假设服务使用者能够完全理解服务规格说明，这样做通常是不明智的。输入消息可能不正确，所以需要定义一个异常，来向服务请求方汇报错误输入。将所有的异常处理留给构件的使用者，这点在可复用构件开发中通常是好的做法。服务开发人员不

应该将他们的观点强加在异常如何处理上。

操 作	输 入	输 出	异 常
MakeCatalog	公司 id PDF- 标志	mcOut 该公司目录的 URL	mcFault 无效的公司 id
Lookup	lookIn 目录的 URL 目录数目	lookOut 含项信息页面的 URL	lookFault 无效目录数目
Search	searchIn 目录 URL 搜索字符串	searchOut 含搜索结果页面的 URL	SearchFault 格式糟糕的搜索字符串 compIn
Compare	目录 URL 入口属性 (最多 6 个) 目录数目 (最多 4 个)	compOut 显示比较表页面的 URL	compFault 无效的公司 id 无效的目录数 未知的属性
CheckDelivery	cdIn 公司 id 目录数目 请求的项数	cdOut 期待的交付日期	cdFault 无效公司 id 无可用性 零请求项
MakeVirtualOrder	voIn 公司 id 目录数目 请求的项数	voOut 目录数目 请求的项数 预计交付日期 单价估计 总价格估计	voFault 无效公司 id 无效目录数目 零请求项

图 18-11 目录接口设计

在某些情况下，只需要关于服务操作以及服务的输入和输出的文本描述。服务的详细实现留作具体实现时再决定。然而有时需要更加详细的设计，可以通过图形表示法（如 UML）或者可读的描述格式（如 JSON）来确定详细的接口描述。图 18-12 展示了如何使用 UML 来详细描述接口，同时给出了操作 `getDelivery` 的输入和输入。

注意本例中如何通过注释带有约束的 UML 图来添加详细描述。这些细节定义了代表公司和目录项的字符串的长度，确定了每个项的数量要大于零，以及交付要在当前日期之后。注释也显示了每个可能的故障关联的错误代码。

目录服务是实际服务的一个实例，说明选择 RESTful 还是基于 SOAP 的方法来实现服务并不总是简单明了的。作为一个基于实体的服务，目录能够表示成资源，这点暗示了 RESTful 模型是正确的选择。然而，对目录的操作并不是简单的 GET 操作，用户需要在与

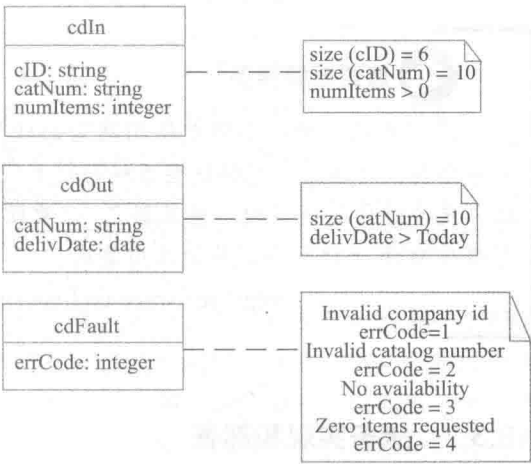


图 18-12 输入、输出消息的 UML 定义

目录服务交互的一次会话中维持一些状态。这点显示了应该使用基于 SOAP 的方法。这种困境在服务工程中是很常见的，并且决定使用哪种方法的关键因素通常是本地环境（如专门技术的可用性）。

为了实现一套 RESTful 服务，必须决定将要用于表示目录的一套资源有哪些，以及基本的 GET、POST 和 PUT 怎么操作这些资源。这些设计决策中的一部分是简单明了的：

1. 应该有一种资源表示特定公司的目录。这种资源的 URL 的格式为：<base catalog>/<company name>，并且应该使用 POST 操作来创建。
2. 每个目录项应该有自己的 URL，格式为：<base catalog>/<company name>/<item identifier>。
3. 使用 GET 操作来寻回目录项。通过将目录中一个项的 URL 用作 GET 操作的参数，可以实现查询。通过将公司目录作为 GET 操作的 URL，以及搜索字符串作为 GET 操作的查询参数，可以实现搜索。这个 GET 操作返回一系列匹配这次搜索的目录项的 URL。

然而，操作 Compare、CheckDelivery 和 MakeVirtualOrder 更加复杂。

1. Compare 操作可以按如下方式实现：一系列的 GET 操作来寻回各个目录项，紧接着通过一个 POST 操作来创建一张对照表，最终一个 GET 操作返回这个对照表给用户。

2. CheckDelivery 操作和 MakeVirtualOrder 操作需要额外的资源来表示一个虚拟的订单。使用 POST 操作来创建资源，并且对应所需要的项数量。使用公司的 ID 来自动填写订单，并且计算出交付日期。接下来就可以使用 GET 操作来寻回资源了。

读者需要仔细思考如何将异常映射到标准 http 响应代码上，如代码 404，这意味着一个 URL 不能被检索。本书没有足够的篇幅来探讨这个问题，但它进一步增加了服务接口设计的复杂度。

对于基于 SOAP 的服务来说，在这个例子中，因为逻辑接口设计可以被自动翻译为 WSDL，所以实现过程更加简单。大部分支持面向服务开发的编程环境（如 ECLIPSE 环境），包含了能够将逻辑接口描述转变为对应的 WSDL 表示的工具。



遗留系统服务

遗留系统是由一个组织使用的老的软件系统。重写或替换这些系统从成本效益上看并不合算，许多组织想将这些系统与更加现代化的系统一起使用。最重要的服务使用方式之一是为遗留系统实现“包装器”，以便提供对系统功能和数据的访问。这样这些系统就可以在 Web 上访问并与其他应用集成。

<http://software-engineering-book.com/web/legacy-services>

18.3.3 服务实现和部署

一旦找到了可选服务并且设计了它们的接口，服务工程过程的最后阶段就是服务实现。实现可能涉及使用某个标准的编程语言（例如 Java 或 C#）编写服务程序。这两种语言现在都包括对基于 SOAP 和 RESTful 的服务的开发提供广泛支持的库。

另外一种做法是，服务的开发可以通过向现有构件或遗留系统（稍后将要讨论）添加服务接口的办法实现。这意味着已经证明了是有用的软件资产能被更加广泛地利用。对于遗留

系统的情形，它可能意味着系统功能能被新的应用访问。也可以通过定义现有服务的组合来开发新的服务。18.4 节讨论这种开发服务的方法。

服务一经实现，在部署它之前必须通过测试。这包括检查和划分服务输入（如第 18 章所讨论的），创建反映这些输入组合的输入消息，然后检查输出是否符合预期。在测试中应该总是去尝试产生异常来检查服务是否能够应付无效输入。测试工具都允许对服务进行检查和测试，并能从 WSDL 描述生成测试。然而，这些只能测试服务接口与 WSDL 的一致性；它们不能测试服务的功能性行为。

服务部署是过程的最后阶段，包括在 Web 服务器上部署此服务。绝大多数服务器软件可以使这步变得非常简单——只需在特定目录下安装包含可执行的服务的文件，然后它会自动变得可用。

如果希望服务在大型机构内是可用的或者是公用的，必须为服务的外部用户提供信息。这些信息帮助潜在的外部用户明确该服务能否满足他们的需求以及是否值得信任，而对于服务提供商，能够帮助他们安全可靠地提供服务。服务描述中可能包含的信息如下。

1. 有关企业的信息、联系方式等。这对用户信任来说是至关重要的。服务的外部用户必须确信服务将不会表现出恶意的行为。有关服务提供者的信息能让用户在商业信息机构中去检查他们的资格证书。

2. 服务提供的功能的非正式描述。这帮助潜在用户决定服务是否是他们想要的。

3. 如何使用服务的描述。对一些简单的服务来说，这将对服务的输入、输出的非正式文字描述。对于更加复杂的基于 SOAP 的服务来说，需要 WSDL 描述。

4. 订阅信息，允许用户注册以获取有关服务更新的信息。

如前面所述，服务规格说明的一个一般问题是，服务的功能行为是通过自然语言描述非正式给出的。自然语言描述便于阅读，但是容易引起误解。为了解决这个问题，在使用本体论和本体语言定义服务语义方面开展了广泛的研究工作（W3C 2012）。然而，基于本体论的规格说明复杂并且难以理解。因此，基于本体论的描述没有被广泛使用。

18.4 服务组合

面向服务的软件开发大概是基于这样的思想：组合并配置服务来创建新的复合服务。这些复合服务可以与一个在浏览器上实现的用户接口集成来创建一个 Web 应用，或者可以被当作构件用于某个其他服务组合。组合中所包含的服务可能是专门为一特殊应用开发的，可能是公司内部开发的业务服务，或者可能是来自某个外部供应商的服务。RESTful 和基于 SOAP 的服务都能够被组合，从而开发出具有扩展功能的服务。

许多公司目前正把他们的企业应用转换为面向服务的系统，其中构建块的基本应用程序是一个服务而不是一个构件。这开启了在公司内部更加广泛复用的可能性。下一个阶段将会是开发机构间的相互信任的供应商之间的应用，它们彼此使用对方的服务。面向服务体系结构的远景目标的最终实现将依赖于“服务市场”的开发，其中的服务均来自可信赖的外部供应商。

服务组合是一个集成过程，可以用来集成分离的业务过程，从而能提供更加广泛的功能。比如，一家航空公司希望为旅客提供一个完整的假期计划。除了预订他们的班机外，旅客也能预订在他们所希望的地点的旅馆，安排租车或预订出租车到机场接机，浏览旅行指南以及预约当地名胜的游览。为了创建此应用，航空公司将它的订票服务、旅馆预订代理所提

供的服务、汽车租赁和出租车公司的服务，以及当地景点提供者所提供的预约服务组合在一起。最终结果是一个服务，它集成了这些来自于不同提供者的服务。

你可以将此过程看作是一些分离步骤的序列，如图 18-13 所示。信息从一步传递到下一步，例如，汽车租赁公司被告知班机预定到达的时间。步骤序列被称为一个工作流——一组在时间上有顺序的活动，每个活动执行工作的某个部分。工作流是业务过程的一个模型，即列出在达到一个重要业务目标过程中所涉及的所有步骤。在本例中，业务过程是航空公司提供的假期预订服务。

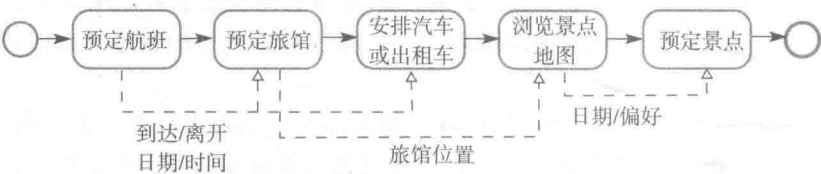


图 18-13 一揽子假期工作流

工作流是一个简单的思想，上述的假期预订的假定情节看起来是很简单的。事实上，服务组合要比这个简单模型所暗含的要复杂得多。举例来说，你必须考虑服务失效的可能性，并加入一些机制来处理这些失效。你也需要考虑应用用户会提出的非预期的要求。举例来说，假定一位旅客是残疾人，需要租用一个轮椅到飞机场。这将需要实现并组合额外的服务，还需要向工作流中添加额外的步骤。

我们必须能够应付这样的局面：工作流不得不改变，因为某一个服务的正常执行总是导致与其他服务执行的不兼容。举例来说，假定所预订的一架班机将在 6 月 1 日离开 6 月 7 日返回。于是工作流进入旅馆预订阶段。然而，旅游景点直到 6 月 2 日一直都在召开一个重要会议，因此旅馆没有可用的房间。旅馆预订服务报告此不可用。这不是一个失效，缺乏可用性是一种常见的情形。

于是我们不得不“撤销”班机预订并将有关缺乏可用性的信息返回给用户。然后他就会决定是否改变他们的日程或他们的度假地。在工作流术语中，这叫作“补偿动作”。补偿动作用来撤销一些动作，这些动作是已经完成的但是因为后续工作流活动的结果必须改变。

通过复用现有服务来设计新服务的过程，本质上是包含复用的软件设计过程（见图 18-14）。包含复用的设计不可避免要在需求上做出妥协。系统“理想化”的需求不得不进行修改以迎合实际上可用的服务，利用这些可用服务，成本就能在预算之内，且服务的品质是可以接受的。

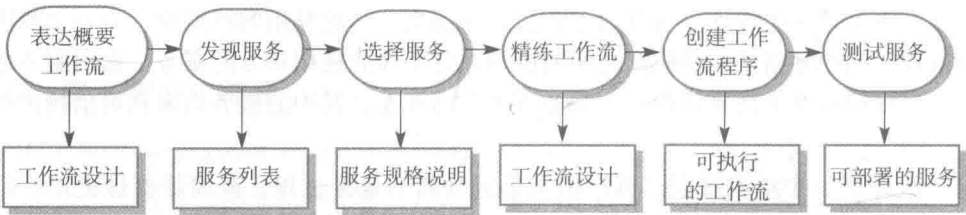


图 18-14 通过组合的服务构造

图 18-14 中，在通过组合构建服务的过程中给出了 6 个关键阶段。

1. 表达概要 workflow。服务设计的起始阶段, 将组合服务的需求作为创建“理想”服务设计的基础。应该在此阶段创建一个相当抽象的设计, 以便在我们对可用服务了解更多的时候能够添加细节。

2. 发现服务。在此阶段, 搜索现有服务以纳入组合中。大部分的服务复用是发生在企业内部的, 所以发现服务应该包括搜索本地服务目录。或者, 应该从一些可信赖的服务提供商 (如 Oracle 和 Microsoft) 提供的服务中搜索。

3. 选择可能的服务。从已经发现的可能的服务候选集合中, 选择能实现 workflow 活动的可能服务。选择标准肯定要包括所提供的服务的功能, 也可能包含所提供服务的成本和服务品质 (响应、可用性等)。

4. 精练 workflow。基于已经选择服务的信息, 精练 workflow。这包括把细节加入到抽象描述中, 可能的话, 还可以增加或删除 workflow 活动。然后可能要重复服务发现和选择阶段。一旦选择了一组稳定的服务, 并建立起了最终的工作流, 就转移到过程的下一个阶段。

5. 创建工作流程序。在此阶段, 将抽象 workflow 设计转换为一个可执行程序, 并定义服务的接口。服务实现可以使用常用的编程语言, 例如 Java 或 C#, 也可以使用工作流语言, 如 BPMN (后文将介绍)。这一阶段也可能包含基于 Web 的用户界面的创建, 从而允许从 Web 浏览器访问新的服务。

6. 测试已完成的服务或应用。在使用外部服务的情形之下, 对完工的组合服务的测试过程比构件测试更为复杂。18.4.2 节将讨论测试问题。

这个过程假设现有的服务可用于服务组合。如果你依赖的外部信息无法从相应的服务接口获取, 你就需要自己实现这些服务。这通常包含一个“屏幕抓取”的过程, 程序需要从浏览器上显示的网页的 HTML 文本中抽取信息。

18.4.1 工作流设计与实现

工作流设计包括分析现有的或计划中的业务过程, 从而理解所发生的不同活动以及这些活动是怎么交换信息的, 接着用工作流设计符号定义新的业务流程。工作流设计列出了执行过程中的各个阶段和不同过程阶段之间所传递的信息。然而, 已有的过程可能是非正式的且依赖于过程中人的技术和能力, 而并没有一个“规范”的工作方式或过程定义。在这种情况下, 我们就不得不使用当前过程的知识来设计工作流, 使之达到的同样的目标。

工作流表示的是业务过程模型, 且通常使用图形形式来描述, 例如, UML 活动图, 或业务过程建模符号 (Business Process Modeling Notation, BPMN) 系统 (White and Miers 2008; OMG 2011)。本章中的例子使用 BPMN 描述。WS-BPEL 是一种基于 XML 的工作流语言, 如果使用基于 SOAP 的服务, 可以将 BPMN 工作流自动转换成 WS-BPEL。这和其他 Web 服务标准 (如 SOAP、WSDL) 相一致。RESTful 服务可以通过类似 Java 的标准编程语言组合到程序中; 或者, 也可以使用用于服务混搭的组合语言 (Rosenberg et al. 2008)。

图 18-15 是 BPMN 模型的一个简单例子, 这是前面的假期计划脚本的部分内容。模型给出了旅馆预订的简化工作流, 假定存在一个 Hotels 服务, 其关联操作称为 GetRequirements、CheckAvailability、ReserveRooms、NoAvailability、ConfirmReservation 和 CancelReservation。此过程包括: 得到来自客户的需求, 然后检查是否有空余房间可用, 如果房间是可用的, 按照所要求的日期预订房间。

此模型引入了 BPMN 的一些用来创建工作流模型的核心概念:

1. 圆角矩形表示活动。一个活动能被一个人或一个自动化服务执行。
2. 圆圈表示事件。一个事件是在一个业务过程当中发生的某事。简单圆圈用来表示一个开始事件，较深的圆圈表示一个结束事件。双层圆圈（没有给出）用来表示一个中间事件。事件可以是时钟事件，因而允许 workflow 定期地或超时地执行。
3. 菱形表示一个通道。通道是过程中的一个阶段，在此做出某个选择。举例来说，在图 18-15 中，有一个选择是根据房间是否可用而做出的。
4. 实线箭头用来表示活动序列；虚线箭头表示活动之间的消息流，在图 18-15 中，这些消息在旅馆预订服务与客户之间传递。

这些主要特征足够描述大多数工作流的本质。然而，BPMN 还包含许多另外的特征，这里不再过多描述。这些特征将信息添加到业务过程描述中，使之能够自动被翻译为一种可执行的服务。

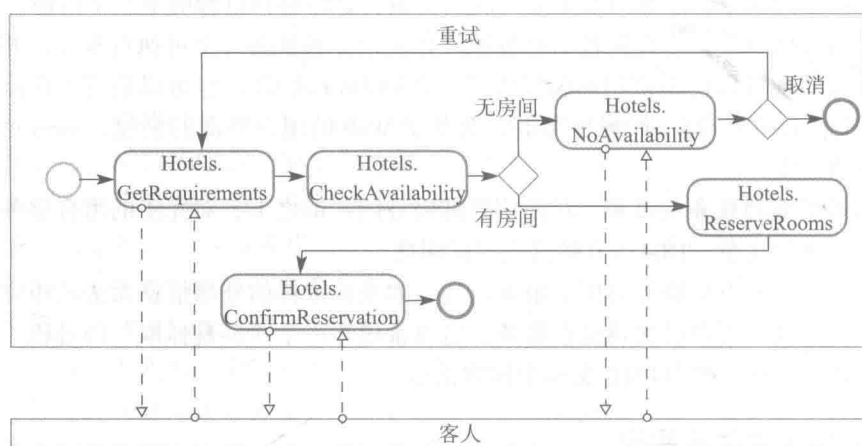


图 18-15 旅馆预订工作流片段

图 18-15 显示的是在某组织中使用的过程，这是一个提供预订服务的公司。然而，面向服务方法的主要好处在于它支持组织间计算。这意味着整个计算涉及不同公司的服务。这可以用 BPMN 表示出来，通过为每个参与交互的组织开发单独的工作流完成的。

为了阐明这一点，采用另外一个例子，该例子与高性能计算有关。面向服务的方法的提出允许共享诸如高性能计算机这样的资源。在本例中，假定一个矢量处理计算机（一个能在值数组上执行并行计算的机器），其作为一个服务（VectorProcService）由一研究实验室提供。该计算机通过另外一个被称为 SetupComputation 的服务进行访问。这些服务以及它们的交互如图 18-16 所示。

在此例中，SetupComputation 服务的工作流请求访问一台矢量处理器，如果有处理器是可用的，就建立所需要的计算并下载数据到正在处理的服务。一旦计算完成，结果便被存储在本地计算机上。VectorProcService 的工作流包括以下步骤：

1. 检查处理器是否可用；
2. 为计算分配资源；
3. 启动系统；
4. 进行计算；
5. 返回结果给客户服务。

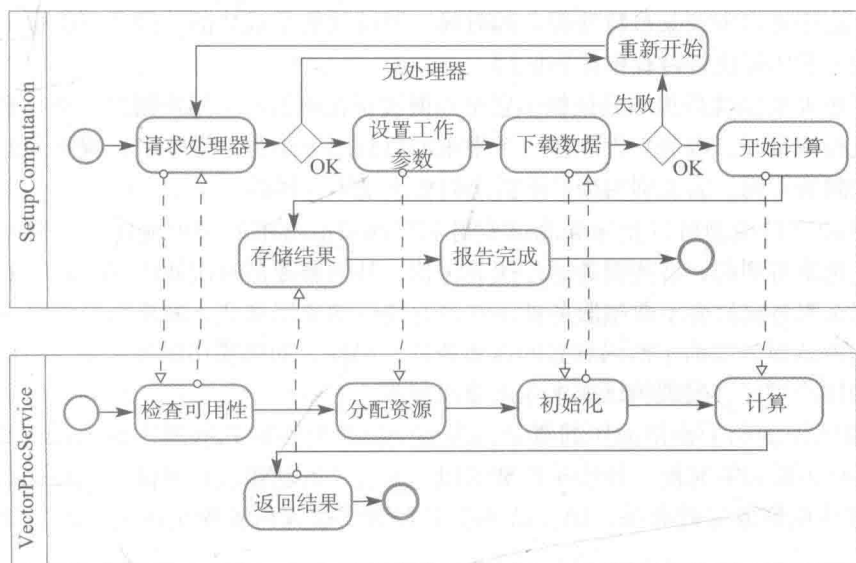


图 18-16 交互的工作流

在 BPMN 术语中，每个组织的工作流被表示在一个独立的池中。用图表示为：将过程中的每个参与者的工作流包在一个矩形中，并在左侧边缘上垂直写下名称。在各个池中定义的工作流通过交换消息完成它们之间的协调。有时候某组织的不同部门参与了一个工作流，此时，池被分隔为一个个“航道”，各个航道表示那个部门中的活动。

一旦业务过程模型设计出来，它必须根据已经发现进行精练的服务。如在对图 18-14 的讨论中所说的，模型可能需要经历很多次重复，直到所做出的设计允许对可用服务进行最多次数的复用。

最终设计一旦就绪,就可以开发最终的面向服务的系统。这包含了实现不可复用的服务,以及将 workflow 模型转换成可执行的程序。因为服务是和实现语言无关的,所以新的服务可以用任何语言来写。如果使用了基于 SOAP 的服务, workflow 模型能够被自动处理,从而创建可执行的 WS-BPEL 模型。如果使用了 RESTful 服务,通过将模型作为程序规格说明,可以手动编写 workflow 模型。

18.4.2 服务组合测试

测试对于所有的系统开发过程来说都是重要的，因为它表明一个系统满足它的功能性和非功能性需求，且检测在开发过程中所导入的缺陷。许多的测试技术，例如程序检查和覆盖测试，依赖于对软件源代码的分析。然而，当服务由一外部提供商所提供时，服务实现的源代码是不可得的。因此不能使用基于系统源代码的“白盒测试”。

在对服务和服务组合进行测试时，除了有对服务实现的理解的问题外，测试者也可能面对更大的困难。

1. 外部服务受控于服务提供商而非服务的用户。服务提供商随时可能撤销这些服务或改变它们，而这些都是将使先前的所有应用测试不可用。这些问题在软件构件中是通过维护构件的不同版本得到解决的，然而现在，还没有提出任何标准去处理服务版本。
2. 如果服务是动态绑定的，那么一个应用在每次执行时可能不总是使用相同的。服务。所

以, 当一个应用被绑定到某个特殊服务的时候, 测试可能是成功的, 但是不能保证那个服务将在系统的一个实际执行过程中得到使用。

3. 服务的非功能性行为不只依赖于它是如何被正在测试的应用使用的。一个服务在测试期间可能执行得很好, 因为它没有在一个很重的负载之下运行。实际中, 观察到的服务行为可能与测试时的不同, 因为别的用户所提出的要求是不一样的。

4. 服务的支付模型可以使服务测试变得非常昂贵。有不同种可能的支付模型——某些服务可以是免费得到的, 有些服务通过注册付款, 其他服务是每次使用时付款。如果服务是免费的, 那么服务提供商不希望服务被正在进行测试的应用加载; 如果需要注册, 那么服务用户可能在测试服务之前不愿同意它的注册条款; 同样, 如果使用服务是基于每次使用付款的, 服务的用户可能会发现测试成本令人望而却步。

5. 前面已经讨论了补偿动作的概念, 当某个异常发生而先前做出的承诺 (例如一次班机预订) 不得不撤销的时候, 补偿动作被调用。对这样的动作进行测试存在一个问题, 因为它们可能与其他服务失效有关。确保这些服务在测试过程内模拟某些服务的失效可能非常困难。

当使用的是外部服务时, 这些问题就特别敏感。当服务来自于同一公司或对伙伴公司所提供的服务信任时, 这些问题就不那么敏感。在这种情况下, 源代码是可以得到的, 可以用它们来指导测试过程, 对服务的支付也不是什么问题。解决这些测试问题, 提出在测试面向服务应用时的准则、工具和技术仍然是一个重要的研究课题。

要点

- 面向服务的体系结构是可复用软件工程的一种方法, 可复用的标准化的服务是应用系统的基本组成部分。
- 通过使用一系列基于 XML 的 Web 服务标准, 可以开发出符合 SOA 的服务。这些 Web 服务标准涉及服务通信、接口定义、工作流中服务的制定。
- 可以使用 RESTful 架构, 这个架构基于资源和这些资源之上的标准操作。一个 RESTful 方法使用 http 和 https 协议来通信, 并且将操作映射到标准 http 的 POST、GET、PUT 和 DELETE 上。
- 服务可以被分为: 提供某一通用目的功能的实用服务, 实现部分业务过程的业务服务, 协调其他服务执行的协同服务。
- 服务工程过程包括: 为实现识别可选服务, 定义服务接口, 实现、测试和部署服务。
- 使用服务的软件开发是基于这样的思想: 程序的创建是通过组合并配置服务来创建复合服务。
- 图形工作流语言 (例如 BPMN) 可以用来描述一个业务过程和业务过程中使用到的服务。这些语言可以描述业务过程中涉及的组织间的交互。

阅读推荐

网上有大量的指导材料, 涵盖了 Web 服务的所有方面。然而, Thomas Erl 所著的下面两本书是对服务以及服务标准最好的综述和阅读材料。与绝大多数的书不同, Erl 在面向服务计算方面给出了一些有关软件工程问题的讨论。他近期还写了一些关于 RESTful 服务的书。

《Service-Oriented Architecture: Concepts, Technology and Design》是一本关于面向服务系统工程的更全面的书，涵盖了 SOA 和 RESTful 体系结构。Erl 讨论了 SOA 和 Web 服务标准，但他主要聚焦在面向服务方法是如何在软件过程的各个阶段使用的。(T. Erl, Prentice Hall, 2005)

《Service-oriented architecture》是一本介绍 SOA 的书，该书很棒并且易读。(多个作者, 2008) <http://msdn.microsoft.com/en-us/library/bb833022.aspx>

《RESTful Web Services:The Basics》是关于 RESTful 方法和 RESTful 服务的一本很好的介绍性的教程。(A.Rodriguez, 2008) <https://www.ibm.com/developerworks/webservices/library/ws-restful/>

《Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL, and RESTful Web Services》是一本面向希望在企业中开发 Web 服务的开发者的进阶教程，描述了很多常见的问题，并且概括了网上对这些问题的解决方案。(R. Daigneau, Addison-Wesley, 2012)

《Web Services Tutorial》是一本关于 SOA、Web 服务和 Web 服务标准的扩展教程，由这些标准的制定者所编写。如果你需要对这些标准有更详细的理解，这本书将非常有用。(W3C schools, 1999-2014) <http://www.w3schools.com/webservices/default.asp>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap18/>

支持视频的链接: <http://software-engineering-book.com/videos/software-reuse/>

练习

- 18.1 服务和软件构件之间的最主要区别是什么?
- 18.2 标准是 SOA 的基石，人们普遍认为标准的一致性对 SOA 被成功采用非常重要。然而，使用越来越广泛的 RESTful 服务并不是基于标准的。请说明这个变化是如何发生的。缺少标准是否会阻碍 RESTful 服务的发展和推广?
- 18.3 使用 WSDL 扩展图 18-5，使其包含对 MaxMinType 和 InDataFault 的定义。温度应该表示为整数，有一个附加字段指示是否为华氏温度或摄氏温度。InDataFault 应该是一个包含一个错误代码的简单类型。
- 18.4 为图 18-7 所示的 Currency Converter(货币兑换)和 Check credit rating(检查信用等级)两个服务分别给出一个接口规格说明。
- 18.5 请说明 CurrencyConverter 服务和 CheckCreditRating 服务如何实现为 RESTful 服务。
- 18.6 请为图 18-13 中的服务设计可能的输入、输出信息。你需要使用 UML 或者 XML 来表示这些信息。
- 18.7 请举出两种不适合使用 SOA 的应用类型，并给出理由。
- 18.8 解释一下“补偿动作”的含义，并用一个例子说明为什么这些动作必须包含在工作流中。
- 18.9 以假期行李预约服务为例，设计一个工作流，能够为刚抵达机场的一批乘客预约交通工具。他们应该能够选择订出租车或者租车。你可以假设出租车公司和汽车租赁公司提供 Web 服务来作预订。
- 18.10 请给出一个例子来详细解释为什么全面测试包含补偿动作的服务很困难。

参考文献

Erl, T. 2004. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, NJ: Prentice-Hall.

———. 2005. *Service-Oriented Architecture: Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice-Hall.

Fielding, R. 2000. "Representational State Transfer." *Architectural Styles and the Design of Network-Based Software Architecture*. https://www.ics.uci.edu/~fielding/pubs/.../fielding_dissertation.pdf

Lovelock, C, S Vandermerwe, and B Lewis. 1996. *Services Marketing*. Englewood Cliffs, NJ.: Prentice-Hall.

Newcomer, E., and G. Lomow. 2005. *Understanding SOA with Web Services*. Boston: Addison-Wesley.

OMG. 2011. "Documents Associated with Business Process Model and Notation (BPMN) Version 2.0." <http://www.omg.org/spec/BPMN/2.0/>

Pautasso, C., O. Zimmermann, and F. Leymann. 2008. "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision." In *Proc. WWW 2008*, 805–14. Beijing, China. doi:10.1145/1367497.1367606.

Richardson, L., and S. Ruby. 2007. *RESTful Web Services*. Sebastopol, CA: O'Reilly Media Inc.

Rosenberg, F., F. Curbera, M. Duftler, and R. Khalaf. 2008. "Composing RESTful Services and Collaborative Workflows: A Lightweight Approach." *IEEE Internet Computing* 12 (5): 24–31. doi:10.1109/MIC.2008.98.

W3C. 2012. "OWL 2 Web Ontology Language." <http://www.w3.org/TR/owl2-overview/>

———. 2013. "Web of Services." <http://www.w3.org/standards/webofservices/>

White, S. A., and D. Miers. 2008. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL. USA: Future Strategies Inc.

系统工程

目标

本章的目标是解释软件工程师为什么要理解系统工程，并介绍最重要的系统工程过程。阅读完本章后，你将：

- 了解社会技术系统的含义并理解为什么人、社会和组织因素会影响软件系统的需求和设计；
- 理解概念化设计的思想以及它为什么会成为系统工程过程中第一个重要的阶段；
- 了解系统采购的含义并理解为什么不同类型的系统会采用不同的系统采购过程；
- 了解关键的系统工程开发过程及关系。

计算机只有当其既包含软件又包含硬件时才会有用。没有硬件，软件系统只是一种抽象，一种关于人类知识和思想的表示。没有软件，硬件系统只是一组冷冰冰的电子设备。然而，如果你把软件和硬件放到一起构成一个计算机系统，那么你就创建了一个可以执行复杂计算并向外部环境提供计算结果的机器。

这展示了一个系统的根本特性：系统不仅仅是它的组成部分的总和。系统具有的一些特性只有当它的所有构件被集成在一起并运行时才会表现出来。此外，系统被开发用来支持人类活动，例如工作、娱乐、交流、保护人类和环境等。系统与人进行交互，系统的设计受人和组织关注点的影响。所有的专业软件系统在开发的时候都必须将硬件、人、社会和组织因素考虑进来。

包含软件的系统可以分为以下两类。

1. 基于计算机的技术系统。这类系统包含硬件和软件构件但不包含相关的规程和过程。这类系统的例子包括电视、移动电话以及其他包含嵌入式软件的设备。个人电脑应用、电脑游戏和移动设备也是技术系统。个人和组织出于特定目的使用技术系统，但关于该目的的知识并不是技术系统的一部分。例如，我所使用的字处理软件（微软的 Word）并不知道它被用于写这本书。

2. 社会技术系统。这类系统包含一个或多个技术系统，但更重要的是这类系统自身还包含理解系统使用目的的人在其中。社会技术系统有定义好的操作和运行过程，而人（操作者）是系统的固有组成部分。这类系统受制于组织策略和规则，还可能会受到外部约束（例如国家法律、法规和政策）的影响。例如，本书是经过一个社会技术性的出版系统创造的，该系统包括多个过程（创作、编辑、排版等）和技术系统（微软 Word 和 Excel、Adobe Illustrator、InDesign 等）。

系统工程（White et al. 1993；Stevens et al. 1998；Thayer 2002）是在考虑系统中的硬件、软件和人的元素的特性的情况下设计整个系统的活动。系统工程包括与采购、规约、开发、部署、运行和维护技术和社会技术系统相关的一切。系统工程师必须考虑硬件和软件的能力以及系统与用户和环境的交互。他们必须考虑系统的服务、系统构造和运行需要满足的约

束，以及系统的使用方式。

在本章中，主要关注大型、复杂的软件密集型系统。这些系统是“企业系统”，即用于实现大型组织目标的系统。企业系统用于政府和军事，以及大型企业和其他公共机构的服务。这些系统是受组织的运行方式以及国家和国际规则和规定的影响的社会技术系统。这类系统可以由一些独立的系统组成，是具有大规模数据库的分布式系统。这些系统有很长的生存周期，并对组织的正常运行具有关键的作用。

出于以下两个原因，软件工程师了解系统工程并积极参与系统工程过程是很重要的。

1. 软件在现在所有的企业系统中都是决定性的要素，而组织中很多高层决策者对于软件的理解有限。软件工程师必须在高层的系统决策中扮演更加积极的角色，以确保系统中的软件是可依赖的并且能够按时按预算完成开发。
2. 作为软件工程师，如果你对于软件如何与其他硬件和软件系统交互有全面的了解，并且对影响软件使用方式的人、社会和组织因素具有更广阔的了解，那么是很有帮助的。这些知识帮助你理解软件的局限性，从而设计出更好的软件系统。

在大型复杂系统的生命期中存在四个相互重叠的阶段（见图 19-1）。

1. 概念设计。这一初始的系统工程活动开发所需要的系统类型的概念。该活动以非技术性的语言说明系统的目的、为什么需要该系统以及用户期望看到的系统高层特性。该活动还可以描述大致的系统约束，例如与其他系统的互操作要求。这些约束限制了系统工程师在设计和开发系统时的自由度。

2. 采购或获取。在这个阶段，概念设计进一步得到完善以提供做出关于系统开发合同决策所需的信息。其中可能涉及关于系统功能在硬件、软件和运行过程等元素之间的分布的决策。你还会决定需要获取哪些硬件和软件、应当选择哪些供应商开发该系统，以及供应合同的条款和条件。

3. 开发。此阶段对系统进行开发。开发过程包括需求定义、系统设计、硬件和软件工程、系统集成、测试。此阶段还会定义系统的运行过程，并设计针对系统用户的培训课程。

4. 运行。此阶段对系统进行部署，对用户进行培训，并将系统投入运行。事先规划的运行过程通常必须进行调整以适应系统工作所处的真实运行环境。此后，系统随着新需求的出现而发生演化。最终，系统的价值逐渐下降，退出使用并被替换。

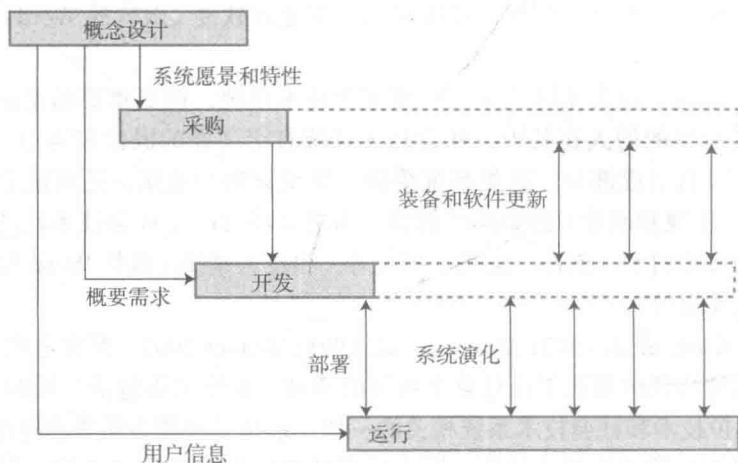


图 19-1 系统工程的各个阶段

图 19-1 描述了这些阶段之间的交互。概念设计活动是系统采购和开发的基础,但是也用于为用户提供系统信息。开发和采购存在重叠,开发和运行过程中当有了新的设备和软件后还可能需要进行进一步的采购。系统进入可运行状态后,需求变更总是不可避免的;实现这些变更需要进一步的开发,还可能需要进行进一步的软件和硬件采购。

在任一阶段所做的决策都可能对其他阶段造成深远的影响。采购决策可能会限制系统的设计选项,包括系统的范围及其软件和硬件组成部分。在规格说明、设计和开发阶段人所犯的错误可能意味着缺陷(故障)被引入到了系统中。因为预算原因限制测试的决定可能意味着缺陷在系统投入使用之前未被发现。在运行过程中,系统部署配置中所犯的错误可能导致系统使用过程中的错误。当系统变更要求被提出时,在最初的采购过程中所做的决策可能被忘掉。这可能导致这些变更的实现产生不可预见的后果。

系统工程和软件工程的一个重要的区别是贯穿整个系统生存期的一系列专业人员的参与。其中包括参与硬件和软件设计的工程师、系统最终用户、关注组织问题的管理人员,以及系统应用领域中的专家。例如,第 1 章所介绍的胰岛素泵系统的系统工程中需要电子、机械工程、软件和医药方面的专家参与。

一些大规模系统还可能涉及更大范围内的专业知识。图 19-2 描述了一个用于空中交通管理的新系统的采购和开发中可能涉及的技术领域。其中包括建筑师和土木工程师,因为新的空中交通管理系统通常都要安装在一座新的建筑中。此外还包括电气和机械工程师,需要他们来设计并维护电力和空调系统。电子工程师关心计算机、雷达和其他设备。人类工程学专家设计控制器工位,软件工程师和用户界面设计师负责系统中的软件。

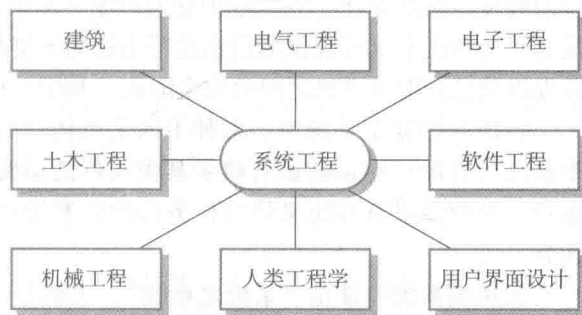


图 19-2 空中交通管理系统工程所涉及的专业领域

在系统工程中综合运用多种专业知识是很重要的,因为复杂系统中包含各种不同类型的构件。然而,不同专业领域之间的差异和误解可能导致不正确的设计决策。这些不正确的决策可能会导致系统的开发延迟或者使系统的满意度下降。具备不同背景的工程师之间存在误解或其他差异主要是因为以下 3 个原因。

1. 不同的专业领域经常使用相同的词汇,但是这些词汇却不总是具有相同的含义。因此,具有不同背景的工程师之间讨论时经常会出现误解。如果这些误解在系统开发过程中没有发现和解决,那么可能导致交付后的系统中存在错误。例如,一个电子工程师可能了解一点 C 程序设计但可能并不理解 Java 中的方法类似于 C 中的函数。

2. 每个专业领域都会对其他领域能或不能做什么做出假设。这些假设经常基于一些不正确的理解。例如,一个电子工程师决定所有的信号处理(一个计算密集型任务)应当由软件来完成以简化硬件设计。然而,这一决定意味着需要极其巨大的软件开发工作量来保证系统处理器可以处理所需的计算量。

3. 每个领域都会尽力保护自身的专业边界,而且会尽力争取系统做出某些特定的设计决策,从而使他们的专业经验和知识有用武之地。例如,软件工程师可能会争取在一栋建筑中采用基于软件的门锁系统,而机械工程则会主张采用基于钥匙的系统会更可靠。

通常，跨专业领域的工作只有在有足够的时间探讨和解决这些问题的情况下才会成功。这需要常规的面对面的讨论以及来自系统工程过程中所涉及的每一个人的灵活方法。

19.1 社会技术系统

系统一词被广泛使用。例如，我们会说计算机系统、操作系统、支付系统、教育系统、政府系统等。这些明显都是关于“系统”的各不相同的使用方式，虽然其中关于系统的本质特性都是一样的，即系统并不只是系统各个组成部分的简单加和。

抽象系统，例如政府系统，不在本书的讨论范围之内。这里关注的是包含计算机和软件并且具有特定的目的（例如，实现通信、支持导航，或者维护医疗记录）的系统。这类系统的一个有用的定义如下：

一个系统是一个相互联系的不同类型的构件的一种有目的的组合，这些构件一起工作来为系统的拥有者及其用户提供一组服务。

这一一般化的定义的适用范围非常广泛，覆盖了很多不同类型的系统。例如，激光笔这样的简单的系统提供了指示服务。该系统可能包含一些硬件构件以及存放在只读存储中的小控制程序。与之相比，一个空中交通控制（ATC）系统则包括成千上万的硬件和软件构件以及基于计算机系统所提供的信息进行决策的人类用户。该系统提供一系列服务，包括为飞行员提供信息、保持飞机之间的安全隔离、利用空域等。

在所有的复杂系统中，各种不同系统构件的属性和行为都不可避免地混合在一起。每个系统构件的成功运转都有赖于其他构件的运转。软件只有在处理器处于工作状态时才能运行。处理器只有在定义计算任务的软件系统已经成功安装的情况下才能执行相应的计算任务。

大规模系统经常是“系统之系统”。也就是说这些系统是由一些独立的系统组成的。例如，一个警察指挥和控制系统可能包括一个地理信息系统以提供各种事件位置的详细信息。同样的地理信息系统可以用于其他系统之中，例如物流运输系统、应急指挥控制系统。系统之系统的工程化在软件工程领域是一个越来越重要的话题，将在第 20 章中详细介绍。

正如在第 10 章的分析，大规模系统除了少数情况之外都是社会技术系统。也就是说，这类系统不仅包括软件和硬件，而且包括人、过程和组织策略。社会技术系统是被用于帮助实现某种业务目的的企业系统。这种业务目的可以是增加销售、减少生产过程中的原材料使用、征税、保持空域安全等。由于这些系统处于组织环境之中，它们的采购、开发和使用不可避免地会受到组织的策略、规程及其工作文化的影响。这些系统的用户是受组织的管理方式以及他们与组织内部和外部的其他人之间的交互方式影响的那些人。

社会技术系统以及使用这些系统的组织之间的密切关系意味着定义系统边界经常很困难。组织中不同的人对系统的边界可能会有不同的视角和看法。这一问题的影响很大，因为在定义系统需求时确定系统的边界之内包含以及不包含什么是很重要的。

图 19-3 反映了这个问题。图中将一个社会

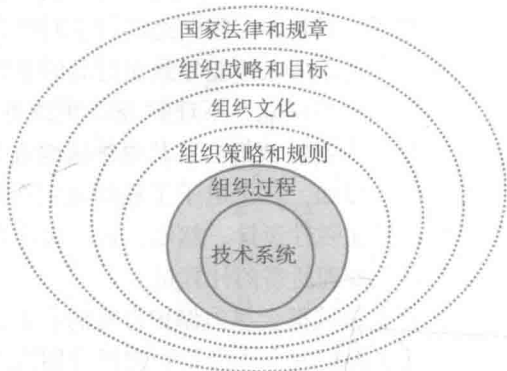


图 19-3 社会技术系统的分层结构

技术系统描述为多个层次,其中每一层都以某种方式为系统的运转做出贡献。处于核心地位的是一个软件密集型技术系统及其运行过程(图19-3中阴影部分)。大部分人都会认同这两层都属于系统的组成部分。然而,系统的行为还会受到核心部分之外的很多社会技术因素的影响。系统的边界是应该简单地画在核心部分周围,还是应该包括其他组织层次呢?

这些更大范围内的社会技术因素是否应该作为系统的组成部分取决于组织及其策略和规则。如果组织规则和策略可以改变,那么有些人会认为它们应该属于系统的组成部分。然而,改变组织的文化则困难得多,而改变组织的战略和目标则更具挑战性。只有政府可以修改法律以容纳一个系统。而且,不同的利益相关者对于系统边界应该画在哪里可能有不同的意见。对于这些问题没有简单的答案,但是这些问题必须在系统设计过程中进行讨论和协商。

一般而言,大型的社会技术系统在组织中使用。当你在设计和开发社会技术系统时,你需要尽可能多地理解系统工作时所处的组织环境。否则,所开发的系统可能会无法满足业务目的,用户及其管理人员可能会抗拒使用该系统或者无法充分利用该系统的能力。

图19-4描述了组织中可能影响一个社会技术系统的需求、设计和运行的关键元素。一个新系统可能会改变这些元素中的一些或全部。

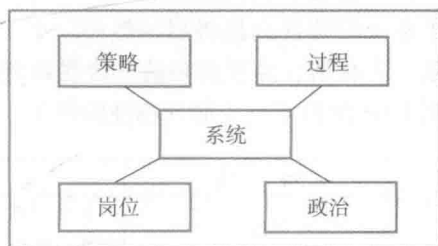


图 19-4 组织元素

1. 过程变化。一个新系统可能意味着人们必须改变自己的工作方式。这样的话还需要对他们进行培训。如果变化十分剧烈或者会导致一些人丢掉工作,那么可能存在用户抵制该系统的引入的风险。

2. 岗位变化。新系统可能使处于相关环境中的用户技能失去用武之地或者导致他们改变自己的工作方式。如果这样的话,用户可能会积极抵制将该系统引入到组织中。专业人员,例如医生或教师,可能会抵制要求他们改变平常工作方式的系统设计。所涉及的人可能会觉得自己的专业技能正在受到侵蚀,他们在组织中的地位正在下降。

3. 组织策略。所规划的系统与组织策略(例如在隐私保护方面)可能并不完全一致。这要求系统、策略或者过程发生变化以使系统和组织策略保持一致。

4. 组织政治。所规划的系统可能会改变组织中的政治权力结构。例如,如果一个组织依赖于一个复杂系统,那么控制对该系统的访问的人将具有很大的权力。反之,如果一个组织将自身重组为另一种结构,那么也会影响系统的需求和使用。

社会技术系统是复杂系统,这意味着在实践中事先取得对系统行为的完整理解几乎是不可能的。这种复杂性导致了社会技术系统以下3个重要特性。

1. 这些系统会涌现出一些整体特性,这些特性是系统的任何一个单独的部分所不具备的。涌现出的整体特性同时取决于系统的构件以及它们之间的关系。这些关系中有一些只有当系统进行构件集成时才会出现,只有到那个时候才能对这些涌现特性进行评价。信息安全和可依赖性是两个重要的系统涌现特性的例子。

2. 这些系统是不确定的,它们接收同样的输入后可以产生不同的输出。系统的行为取决于人类操作者,而人并不总是以同样的方式做出反应。此外,系统在使用过程中还可能会创造新的系统构件之间的关系,从而改变系统的涌现行为。

3. 判断系统成功的准则是主观而不是客观的。系统能够在多大程度上支持组织的目标不

仅取决于系统自身，还取决于组织目标的稳定性、组织目标之间的关系和冲突，以及组织中的人如何理解这些目标。新的管理层可能会重新理解系统设计时所考虑的组织目标，从而导致一个“成功的”系统可能会被认为不再适应于组织的目标。

在确定一个系统是否成功实现其目标时，对社会技术因素的考虑往往十分重要。不幸的是，考虑这些因素对于那些在社会和文化思考方面没什么经验的工程师而言是十分困难的。为了帮助理解系统对于组织的影响，人们提出了一些社会技术系统方面的方法学。本书作者关于社会技术系统设计的论文讨论了这些社会技术系统设计方法的优点和缺点（Baxter and Sommerville 2011）。

19.1.1 涌现特性

一个系统中不同构件之间的复杂关系意味着系统不仅仅是其组成部分的简单加和。系统具有一些整体特性。这些“涌现特性”（Checkland 1981）无法归于系统的任何一个特定的部分，而是当系统构件集成到一起之后才会涌现出来。一些涌现特性，例如重量，可以由子系统的特性直接得出。然而，更多的时候这些特性是由子系统特性以及子系统之间的关系二者相结合涌现出来的。这些系统特性无法直接在单个系统构件的特性基础上计算得到。图 19-5 给出了一些涌现特性的例子。

特 性	描 述
可靠性	系统可靠性取决于构件的可靠性，但非预期的构件间交互可能会导致新的失效类型，从而影响系统的可靠性
可修复性	这一特性反映了发现系统的一个问题之后进行修复的容易程度。该特性取决于诊断问题、访问故障构件以及修改或替换相关构件的能力
信息安全性	系统的信息安全性（系统抵御攻击的能力）是一种无法简单衡量的复杂属性。攻击可能会超出系统设计者所预料的范围，因此可能会攻破系统内置的保护措施
可用性	该属性反映了使用系统的难易程度。该属性取决于技术系统构件、系统操作者以及系统运行环境
体积	系统的体积（所占用的总空间）取决于构件装配是如何安排和连接的

图 19-5 涌现特性的例子

有两种类型的涌现特性。

- 1. 功能性涌现特性，系统的目的仅当其构件集成之后才会表现出来。例如，一辆自行车在其构件组装完成后才会具有成为交通工具的功能性特性。
- 2. 非功能性涌现特性，与系统在其运行环境中的行为相关。可靠性、性能、安全性、信息安全性是这类特性的例子。这些系统特性对于基于计算的系统而言十分重要，因为如果不能达到这些属性所要求的最低级别，那么系统通常会失效。一些用户可能不需要系统的一些功能，因此系统没有这些功能可能也是可以接受的。然而，一个不可靠或太慢的系统很可能会被所有的用户拒绝。

涌现特性，例如可靠性，取决于单个构件的特性，又取决于构件之间的交互或关系。例如，一个社会技术系统的可靠性受以下三方面因素的影响。

- 1. 硬件可靠性。一个硬件构件出现故障的概率如何，以及修复一个故障构件需要多长时间？

2. 软件可靠性。一个软件构件有多大可能性会产生不正确的输出？软件失效与硬件失效的不同之处在于软件不会磨损。失效经常是短暂的。系统在产生不正确的结果后还可以继续运行。

3. 操作人员可靠性。一个系统的操作人员有多大的可能性会犯错误并提供不正确的输入？软件有多大的可能性会检测不到该错误并将错误传播出去？

硬件、软件和操作人员可靠性并不是相互独立的，而是以非预期的方式相互影响的。图 19-6 描述了一个系统中某个层次上的失效是如何传播到其他层次的。例如，一个系统中的某个硬件构件开始出错。硬件失效有时会生成超出软件所预期的输入范围的失真信号。于是，软件可能会以无法预测的方式运行并产生非预期的输出。这些可能会使系统操作人员感到困惑并因此导致他们感到压力。

我们知道人在感到压力的时候更容易犯错。因此一个硬件错误可能会导致操作人员犯错。这些错误又会导致非预期软件行为，从而产生额外的处理器负载。这可能会导致硬件过载，导致更多的失效等。因此，一个初始的较小的失效可能会快速发展为一个可能导致系统完全停机的严重问题。

一个系统的可靠性取决于系统使用所处的上下文环境。然而，我们通常无法完整地刻画系统的运行环境，而且系统设计者通常都无限定系统的运行环境。在同一个环境中运行的不同的系统可能会以无法预测的方式对错误做出不同的反应，从而影响所有这些系统的可靠性。

例如，一个系统是按照通常的室内温度设计的。为了适应不同的以及异常环境，系统的电子构件被设计为可以在一定范围内的不同温度下工作，例如从 0 ~ 40 摄氏度。超出此温度范围的话，这些构件会以无法预测的方式工作。现在假设该系统被安装在一个空调附近。如果这个空调发生失效并且对着这些电子构件吹热风的话，那么系统可能会过热。这些构件以及整个系统都可能会失效。

如果该系统安装在环境中的其他地方，那么这个问题就不会发生了。当空调正常工作的时候，没有问题。然而，由于这些机器在物理位置上过于接近，它们之间产生了一种意想不到的关系，这种关系导致了系统失效。就像可靠性一样，性能或可用性等涌现特性很难评价，但是可以在系统投入运行后进行度量。然而，安全性和信息安全性这样的特性无法直接度量。因为它们并不是仅仅简单地取决于与系统行为相关的属性，而且还与我们不想出现或不可接受的系统行为相关。

一个信息安全的系统不会允许对其数据的非授权访问。不幸的是，事先预测所有可能的非授权访问模式并且显式地禁止它们显然是不可能的。因此，只有在系统投入运行之后才能对这些“不应当”的属性进行评价。也就是说，只是在有人设法侵入系统时才知道系统是不安全的。

19.1.2 不确定性

一个确定性的系统是绝对可预测的。如果我们不考虑并发问题，那么运行在可靠的硬件

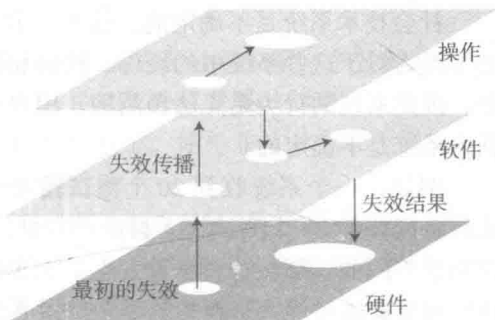


图 19-6 失效传播

之上的软件系统是确定性的。给定一个输入序列，它们总是产生同样的输出序列。当然，并不存在所谓的完全可靠的硬件，但是硬件通常还是足够可靠，因此硬件系统可以被认为是确定性的。

而另一方面，人是不确定的。给定完全一样的输入（即一个任务请求），他们的响应将取决于他们的精神和身体状态、发出请求的人、环境中的其他人，以及他们正在做的其他事情。有时候他们会很高兴完成别人请求的任务，而有的时候他们则会拒绝；有时候他们会完成得很好，而有些时候则完成得很糟糕。

社会技术系统是不确定的，这在一定程度上是因为这些系统包括人，此外也在一定程度上是因为对于这些系统中的硬件、软件和数据的变化过于频繁。这些变化之间的交互十分复杂，因此系统的行为是无法预测的。用户不知道什么时候以及为什么会发生变化，因此他们认为系统是不确定的。

例如，一个系统收到 20 个测试输入。该系统处理这些输入，处理结果会被记录下来。此后，同样的 20 个测试输入再次进行处理，将结果和此前保存的结果进行比较。其中 5 个结果是不同的。那么这意味着发生了 5 次失效吗？或者这些差异只是系统行为的合理变化而已？你只能通过深入分析输出结果并对系统处理每个输入的方式进行推测才有可能做出相应的判断。

不确定性经常被认为是一件坏事，并且设计者应当尽其所能避免系统的不确定性行为。事实上，在社会技术系统中，非确定性有着重要的益处。它意味着一个系统的行为并不总是固定的，而是可以根据系统的环境进行变化。例如，操作人员可能会观察到系统正表现出失效的迹象。此时，他们会改变自己的行为来诊断问题，并从所发现的问题中进行恢复，而不是按照常规方式使用系统。

19.1.3 成功准则

一般而言，复杂的社会技术系统都是被开发用来应对“非常规问题”的（Rittel and Webber 1973）。非常规问题是指问题十分复杂，其中包含相当多的相互关联的实体，以至于无法明确给出问题的规格说明。不同的利益相关者看问题的方式不同，没有人对问题完全理解。问题真正的形态只有当一个解决方案开发出来之后才会浮现出来。非常规问题的一个极端的例子是应对地震救援的应急规划。没有人能准确预测地震的震中在哪里，什么时候会发生，或者对本地环境会造成什么样的影响。详细刻画如何应对该问题是不可能的。系统设计者必须做出假设，但是只有当地震发生后才会知道需要什么。

这使得我们很难定义一个系统的成功准则。如何才能确定一个新系统是否对为此付钱的公司业务目标有贡献？对于一个系统是否成功的判断通常并不是基于采购和开发该系统时的考虑来确定的。这一判断是基于该系统在部署时是否有效而做出的。由于业务环境可能会快速变化，业务目标可能会在系统开发期间发生很大变化。

当存在多个相互冲突、不同利益相关者的理解各不相同的目标时，情况变得更加复杂。例如，Mentcare 系统所基于的系统被设计用来满足两个不同的业务目标：

1. 提高精神病病人诊疗的质量；
2. 通过为管理人员提供关于诊疗服务及其费用的详细报告，提高诊疗的经济性。

不幸的是，这两个目标被证明是冲突的，因为满足报告目标所需的信息意味着医生和护士必须在他们常规记录的健康信息之上以及之外提供更多的信息。这一点降低了为病人的诊

疗效果,因为这意味着医护人员与病人的交谈时间减少了。从一个医生的角度看,该系统与此前的手工系统相比并没有什么改进,但是从管理人员的角度看则有改进。

因此,在系统工程过程早期阶段所建立的任何成功准则通常都必须在系统开发和使用过程中重新进行考虑。你无法对这些准则进行客观评价,因为它们取决于系统对于其环境和用户所产生的效果。一个系统可能会明显满足最初定义的需求,但事实上则可能由于使用环境的变化而变成无用的。

19.2 概念设计

一个系统的最初设想一旦被提出,概念设计就是在系统工程过程中第一个要做的事情。在概念设计阶段,基于这个最初的设想评估其可行性,并对其进行发展以创建一个可以实现的系统总体愿景。接下来,必须以某种方式描述所设想的系统,使得非技术专家,例如系统用户、公司的高层决策者或者政治家,可以理解你所说的。

概念设计与需求工程之间存在着明显的重叠部分。作为概念设计过程的一部分,你必须设想一下所提出的系统将如何被使用。这可能会包含与潜在用户及其他利益相关者的讨论、焦点小组、观察现存系统的使用方式。这些活动的目的是理解用户如何工作、对于他们重要的事情、以及系统可能有哪些现实约束。

建立一个关于所设想系统的愿景的重要性很少在软件设计和需求文献中提及。然而,多年以来这一愿景一直都是军用系统的系统工程过程中的一部分。Fairley 等人 (Fairley, Thayer, and Bjorke 1994) 介绍了概念分析的思想以及如何在“运行概念”(Concept of Operations, ConOps) 文档中描述概念分析的结果。开发一份 ConOps 文档的思想现在已经在大规模系统中得到了广泛使用,你可以在网上找到很多关于 ConOps 文档的例子。

不幸的是,正如许多军用系统和政府系统那样,好的思想也会在僵化并且不灵活的标准中变得一团糟。ConOps 就是这样,有一个这样的文档标准 (IEEE, 2007)。正如 Mostashari 等人 (Mostashari et al. 2012) 所言,这一标准会造成冗长、不可读的文档,无法实现它们最初的目的。他们提出了一种更加敏捷的开发 ConOps 文档的方法,具有更加短小而且更加灵活的文档作为该过程的输出。

我不喜欢“运行概念”(Concept of Operation) 这个词,一定程度上是因为它所具有的军用含义,同时也是因为我觉得概念设计文档不仅仅与系统运行相关,这个文档还应该描述系统工程师关于系统的其他一些理解,包括为什么要开发该系统、关于当前设计方案的合理性解释,有时候还会包括系统的初始组织结构。正如 Fairley 所说,“它应当被写得像讲故事一样”,也就是说写得让没有技术背景的人也能理解其中所提出的方案。

图 19-7 描述了概念设计过程可能包括的活动。概念设计应当都是一个团队过程,其中包括具有不同背景的人。例如,笔者就是第 1 章中所介绍的数字化学习环境的概念设计团队的一员。这个数字化学习环境的设计团队包括教师、教育研究者、软件工程师、系统管理员以及系统管理者。

概念构想是该过程的第一个阶段,在此阶段试图对关于系统目的的初始陈述进行细化,并确定什么样的系统最能够满足系统利益相关者的需要。在开发数字化学习环境时,我们一开始被要求提出一个比现在的系统更容易使用的学校间信息共享的内部网络方案。然而,与教师们讨论后,我们发现这并不真是人们所需要的。当前系统用起来很糟糕,但是人们已经有变通方法了。他们真正需要的是一个灵活的数字化学习环境,能够从互联网上免费获取并

添加课程以及面向不同年龄的工具和内容。

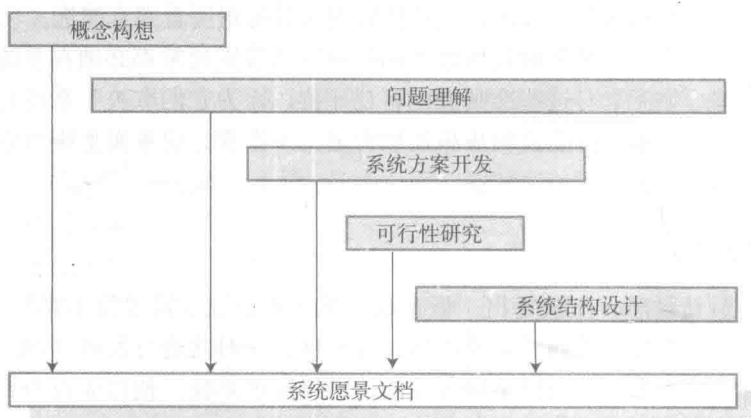


图 19-7 概念设计活动

我们发现这一点是因为概念构想活动与问题理解活动有重叠。为了理解问题，需要与用户和其他利益相关者探讨他们是如何工作的。需要找出哪些东西对于他们是重要的、哪些障碍使得他们无法做自己想做的事情，以及他们对于系统所需要的改变的想法。你需要虚心听取他们的意见（这是他们的问题而不是你的），并且准备在现实与你最初的设想不一致时随时改变自己的思想。

在系统方案开发阶段，概念设计团队拟定一组候选的系统方案，这些设想是可行性研究的基础，而可行性研究将确定哪一个候选方案值得进行进一步的开发。在可行性分析中，你应当留意一些其他地方开发的相似系统以及可能影响系统使用的技术问题（例如移动设备的使用）。接着，你需要评估系统能否使用当前的硬件和软件技术来实现。

还有一个有用的活动是为系统设计一个概要结构或体系结构。该活动对于做出可行性评价以及为更详细的需求工程和体系结构设计提供基础两个方面都是有幫助的。此外，由于当前的大部分系统都是在现有系统和构件基础上装配而成的，因此一个初始的体系结构意味着系统的关键部分都已经识别出来了并且可以分别进行获取。这种方法经常会比以整体的方式从单个供应商那里获取一个单体系统要好。

对于数字化学习环境而言，我们决定使用一种分层服务体系结构（见图 1-8）。系统的所有构件都应当是可替换的服务。通过这种方式，用户可以用他们喜欢的服务来替换标准服务以适应使用该系统进行学习的学生们的不同年龄和兴趣。

所有这些活动都会产生可用于开发系统愿景文档的信息。该文档是一份重要的文档，高层决策者需要使用该文档来确定该系统的进一步开发是否需要进行。该文档还会被用于开发进一步的文档，例如风险分析和预算估计，这些也是决策过程的重要输入。

管理者使用系统愿景文档来理解系统；采购团队使用该文档来定义采购要求文档；需求工程师将文档作为细化系统需求的基础。因为这些不同的人所需要的详细程度不同，建议按照以下两个部分组织该文档。

1. 面向高层决策者的一小段概述，其中陈述了问题的关键要点以及所建议的系统。这一部分应当写得能让读者立刻了解系统将如何被使用以及该系统可以带来的收益。
2. 一些更详细地陈述相关设想的附录，这些内容可以在系统采购和需求工程活动中使用。

编写关于系统愿景的概述是很有挑战性的，因为这部分的读者都很忙并且不太可能具有技术背景。使用用户故事很有效，它可以提供通俗易懂、非技术人员容易接受的关于系统使用的愿景描述。正如图 19-8 所示，故事应当短小并且是个性化的，应当是关于系统使用的一种可行的描述。第 4 章中有另一个来自同一系统的用户故事的例子（见图 4-9）。

数字化艺术

Jill 是敦提（Dundee）的一所中学的学生。她自己有一部智能手机，她家里有一个共用的三星平板电脑和一个戴尔笔记本电脑。在学校的时候，Jill 登录学校的计算机之后可以看到一个个性化的 Glow+ 环境，其中包括一系列服务（有些是她的老师们选择的，有些则是她自己从 Glow 应用库里面选择的）。

她正在做一个凯尔人艺术项目，因此她使用 Google 来研究一些艺术网站。她先在纸上画出一些设计草图，然后使用她的手机上的摄像头把它拍下来，并使用学校的无线网络把照片上传到她的个人 Glow+ 空间中。她的家庭作业是完成这个设计并且把自己的想法写成一段说明。

在家里，她使用家里的平板电脑来登录 Glow+，然后使用一个艺术品应用来处理她的照片并且进一步完善作品、涂色等。她完成这部分工作之后，为了完成作业她转而使用家里的笔记本电脑来撰写自己的说明。她将已完成的作业上传到 Glow+ 上并且发一条消息给一位现在有空对她的作业进行评阅的艺术老师。她的老师在 Jill 的下一堂艺术课之前抽空用学校的平板电脑看了她的项目，然后在课堂上与 Jill 讨论了这个作品。

讨论之后，Jill 和她的老师决定将这个作品分享出去，因此她们将该作品发布到了学校进行学生作品展示的网页上。此外，这个作品也放在了 Jill 的电子书包中，这里面将保存她从 3 岁到 18 岁的学校作业。

图 19-8 一个系统愿景文档中所使用的用户故事

用户故事很有效，因为正如前面所讨论的，读者容易接受用户故事；此外，用户故事可以以一种容易理解的方式显示所建议的系统的功能。当然，这些只是系统愿景的一部分，系统概述还必须包括所做出的基本假设的高层描述以及系统将以何种方式为组织创造价值。

19.3 系统采购

系统采购或系统获取是一个以从系统供应商那里购买一个或多个系统的决策为结果的过程。在这个阶段，决策是根据需要购买的系统的范围、系统预算和时间要求、高层系统需求等做出的。在这些信息基础上，将做出关于是否采购一个系统、所需要的系统类型、系统的供应商等方面的进一步决策。这些决策的驱动力包括以下几个方面。

1. 组织中需要替换其他系统。如果该组织混合了一些无法一起工作或者维护成本昂贵的系统，那么采购一个具有新能力的替换系统可能会带来显著的业务效益。
2. 符合外部监管的需要。一个组织的业务受到的监管越多，就越需要展现与外部监管要求的符合性（例如美国的萨班斯-奥克斯利会计准则）。这一符合性可能会要求替换不相符的系统或者提供新系统来专门对系统的符合性进行监测。
3. 外部竞争。如果某个业务需要获得或者保持竞争地位，那么管理者可能会决定购买新的系统来提高业务效率和有效性。对于军用系统而言，提高面对新威胁时的应对能力是采购新系统的一个重要理由。
4. 业务重组。业务以及组织中的其他部分经常为了改进效率和 / 或客户服务而重组。重组导致业务过程的变更，从而要求新的系统支持。
5. 可用的预算。可用的预算显然是决定可以采购的新系统范围的一个因素。

此外，采购新的政府系统经常可以反映政治政策方面的变化。例如，政治家可能会决定购买他们声称可以反恐的新的监控系统。政治家们通过购买这种系统向选民们表明他们在采取行动。

大型复杂系统的开发通常都会同时采用商用第三方成品构件以及专门构造的构件。这些构件经常要与已有的遗留系统和组织的数据库相集成。当使用遗留系统和第三方成品系统时，可能需要新的定制化软件来集成这些构件。新的软件管理构件接口从而使它们可以互操作。开发这种“黏合剂”的需要在一定程度上使得使用第三方成品构件带来的节省有时没有预期那么大。

有 3 种类型的系统或系统构件可能是必须采购的：

1. 无须修改就可以使用的第三方成品应用，通常只需要按照使用要求进行少量的配置；
2. 必须通过修改代码或者使用内建的配置特性（例如，过程定义和规则）进行修改或适配才能使用的可配置应用或 ERP 系统；
3. 必须特别设计和实现才能使用的定制化系统。

这些构件所遵循的采购过程往往各不相同。图 19-9 反映了这 3 类系统采购过程的主要特征。采购过程之后的关键问题如下。

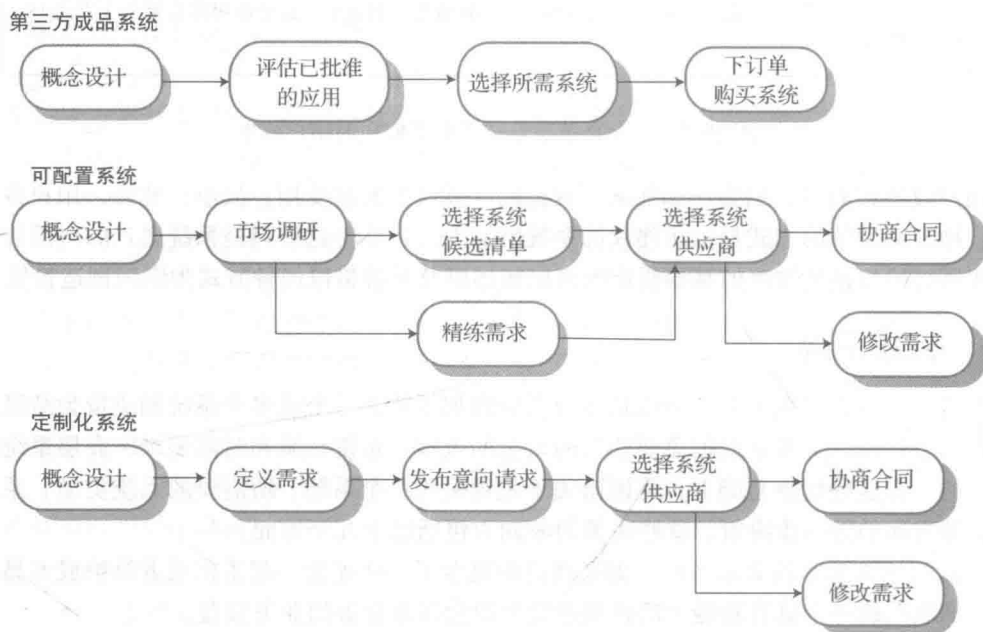


图 19-9 系统采购过程

1. 组织经常有一组已经由 IT 部门检查过的、已批准和推荐的应用软件清单。通常有可能无须论证直接从该清单中购买或获取开源软件。例如，在数字化学习系统中，我们推荐使用 Wordpress 来为学生和教师提供博客服务。如果需要手机，那么可以购买第三方成品硬件。这里没有详细的需求，用户需要适应所选择的应用所提供的特性。

2. 第三方成品构件通常并不是完全与需求匹配的，除非需求是在考虑这些构件的情况下编写的。因此，选择一个系统意味着你必须找到系统需求与第三方成品系统所提供的特性之间的最佳匹配。ERP 和其他大规模应用系统通常属于这一类。你可能不得不修改需求以与系

统假设相适应。这可能会对其他子系统产生影响。你通常还需要一个广泛的配置过程,根据购买方的工作环境对应用或 ERP 系统进行裁剪和适配。

3. 当一个系统需要专门构建时,需求规格说明是所获取的系统的合同的一部分。因此,该需求规格说明既是一份法律文书同时也是一份技术文档。需求文档很重要,这种类型的采购过程通常需要花费很长时间。

4. 对于公共部门的系统而言,存在影响系统采购的详细规则 and 规定。例如,在欧盟,所有的公共部门系统如果超过一定的价格那么必须公开面向欧洲的任何供应商进行招标。这需要草拟详细的招标文档,并且招标要面向整个欧洲进行一定时间的公开宣传。这一规定不仅减慢了招标过程,而且还存在不利于敏捷开发的倾向。它迫使系统购买方明确定义需求使得所有公司都能获得足够的信息来对系统进行投标。

5. 对于需要修改的应用系统或者定制化系统,通常都会有一个合同协商阶段,使得客户和供应商可以就系统开发的条款和条件进行协商。一旦确定选择某个系统,你可以与供应商就成本、许可证条件、对于系统的可能的修改以及其他合同条款进行协商。对于定制化系统,协商的内容很可能会包括支付规划、报告、验收准则、需求变更要求、系统变更成本等。在此过程中,双方可能会就一些需求变更达成一致以降低整体成本同时避免一些开发问题。

复杂的社会技术系统很少由系统的购买方“关门”进行开发。他们往往会邀请外部的系统公司参与系统工程合同的投标。客户的业务并不是系统工程,因此他们的雇员自身并没有自己开发系统所需的技能。对于复杂的硬件/软件系统,有可能要使用一组供应商,其中每一个都具有不同类型的知识和经验。对于大型系统,例如一个空中交通管理系统,一组供应商可以组成一个联合体来对一个合同进行投标。该联合体应当包括这类系统所需的所有能力。对于空中交通管理系统而言,该联合体将包括计算机硬件供应商、软件公司、外围供应商以及专业设备(例如雷达系统)供应商。

客户通常并不希望与多个供应商进行协商,因此合同通常都会与一个主承包商签订,由他们对项目进行协调。主承包商对不同子承包商各自负责的子系统的开发进行协调。子承包商按照与主承包商和客户协商一致的规格说明对系统的一些部分进行设计和构造。一旦完成,主承包商会将这些构件集成起来并将它们交付给客户。

在系统工程过程的采购阶段做出的决策对于该过程此后那些阶段而言十分重要。糟糕的采购决策经常导致各种问题,例如,系统的延迟交付,开发出的系统不适合其运行环境等等。如果选择了错误的系统或错误的供应商,那么系统和软件工程的技术过程将变得更加复杂。

例如,笔者研究过一个系统“失效”,其中所包含的一个决策选择了一个 ERP 系统,因为这将使得整个组织的运转能够“标准化”。但实际上该组织内的业务运行方式多种多样,事实证明这样有很多好处。标准化实际上是做不到的。该 ERP 系统无法适应这种多样性。该系统最终在造成大约 1 000 万欧元的开销之后被抛弃了。

系统采购阶段所做的决策和选择,对于系统的信息安全和可依赖性有着深远的影响。例如,如果决定采购一个第三方成品系统,那么该组织必须接受他们对该系统的信息安全和可依赖性需求毫无发言权的事实。系统信息安全性取决于系统供应商所做的决策。此外,第三方成品系统可能会有已知的信息安全弱点,或者可能需要复杂的配置。配置错误可能导致系统的入口没有得到妥善的保护,这是信息安全问题的一类突出的原因。

另一方面, 采购一个定制化系统的决策意味着需要花费很大的工作量来理解和定义系统的信息安全和可依赖性需求。如果一个公司在这个领域经验不足, 那么这将是很困难的一件事情。如果要达到所要求的可依赖性等级以及可接受的系统性能, 那么开发时间将不得不延长而且预算将会增加。

很多糟糕的采购决策源于政治而非技术因素。高级管理层可能希望拥有更多的控制权, 因此要求整个组织都使用一个单一的系统。供应商选择可能是因为他们与公司的长期关系而非他们能够提供最好的技术。管理者可能希望保持与已有系统的兼容性, 因为他们感受到了新技术的威胁。正如第 20 章所讨论的, 不理解所需要的系统的人反而经常负责采购决策。工程问题并不一定在他们的决策过程中扮演主要的角色。

19.4 系统开发

系统开发是一个复杂的过程, 在此过程中系统的各部分元素分别被开发或购买, 然后被集成到一起创造出最终的系统。系统需求是概念设计和开发过程之间的桥梁。概念设计过程中对业务和高层功能性及非功能性系统需求进行了定义。你可以认为这是开发的开始, 因此如图 19-1 所示, 这些过程之间存在重叠。一旦针对系统元素的合同已经达成一致, 就要进行更加详细的需求工程。

图 19-10 是一个系统开发过程模型。系统工程过程通常遵循一种与第 2 章所介绍的“瀑布”过程模型相似。虽然瀑布模型对于大部分类型的软件开发都是不适合的, 但是更高层次上的系统工程过程仍然是计划驱动的, 因此仍然可以遵循这一模型。

系统工程中采用计划驱动的过程, 因为系统的不同元素是独立开发的。不同的承包商同时各个独立的子系统上进行工作。因此, 与这些元素之间的接口必须在开发开始之前设计好。对于包含硬件和其他设备的系统, 开发过程中的变更可能会非常昂贵, 甚至在实践中有可能无法进行任何变更。因此, 在硬件开发或构造开始之前充分理解系统需求是非常重要的。

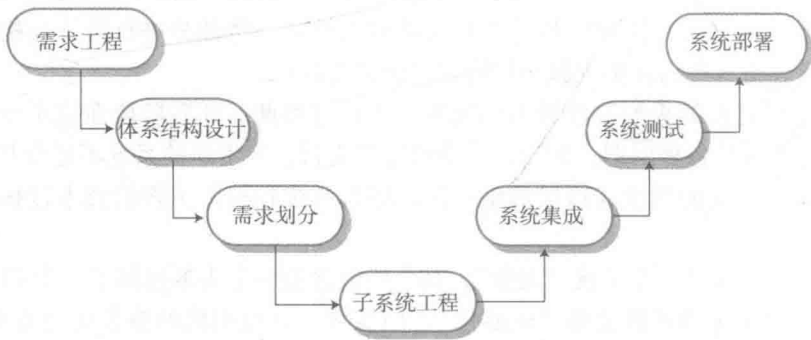


图 19-10 系统开发过程

系统工程中最令人困惑的方面之一就是各家公司使用不同的术语来称呼系统工程过程的各个阶段。有时, 需求工程是开发过程的一部分, 而有的时候需求工程又是一个独立的活。然而, 在概念设计之后有几个基本的开发活动。

1. 需求工程, 是一个精练的过程, 即对概念设计中所识别出的高层和业务需求进行细化、分析和文档化描述。第 4 章中介绍了最重要的需求工程活动。

2. 体系结构设计，与需求工程过程显著存在重叠。该过程包括建立系统的总体体系结构、识别不同的系统构件并理解构件之间的关系。

3. 需求划分，关注确定哪个子系统（在系统体系结构中所识别的哪些子系统）负责实现系统需求。需求可能必须分配给硬件、软件或运行过程，并确定需求实现的优先级。理想情况下，应该为各个子系统分配需求，使得每个重要需求的实现都不需要子系统之间的协作。然而，这并不总是可行的。在此阶段，还会确定运行过程以及这些运行过程如何在需求实现中使用。

4. 子系统工程，包括开发系统的软件构件、配置成品硬件和软件，（如果需要）设计特定目的的硬件、定义系统的运行过程、重新设计基本的业务过程。

5. 系统集成，是一个将系统元素放到一起来创建一个新系统的过程。只有到此时系统的涌现特性才会变得明显。

6. 系统测试，是一个延伸的活动，在此过程中对整个系统进行测试，从而使问题被暴露出来。在此过程中，可能会重新进入子系统工程和系统集成阶段以修复所发现的问题、对系统性能进行调优，以及实现新需求。系统测试既包括系统开发者进行的测试又包括采购该系统的组织进行的验收 / 用户测试。

7. 系统部署，使得系统能够被用户所使用，从现有的系统中进行数据转换，并且与环境中的其他系统建立通信关系。该过程在系统上线后结束，此后用户开始使用系统来支持自己的工作。

虽然总体过程是规划驱动的，需求开发和系统设计过程却不可避免地联系到了一起。需求和高层设计是同时开发的。现有系统的限制可能会限制设计的选择，这些选择可以在需求中进行陈述。可能要做一些初始的设计来对需求工程过程进行构造和组织。随着设计过程继续进行，可能会发现现有需求中的问题，而新的需求也有可能出现。因此，可以将这些相互联系的过程视为一种螺旋结构，如图 19-11 所示。

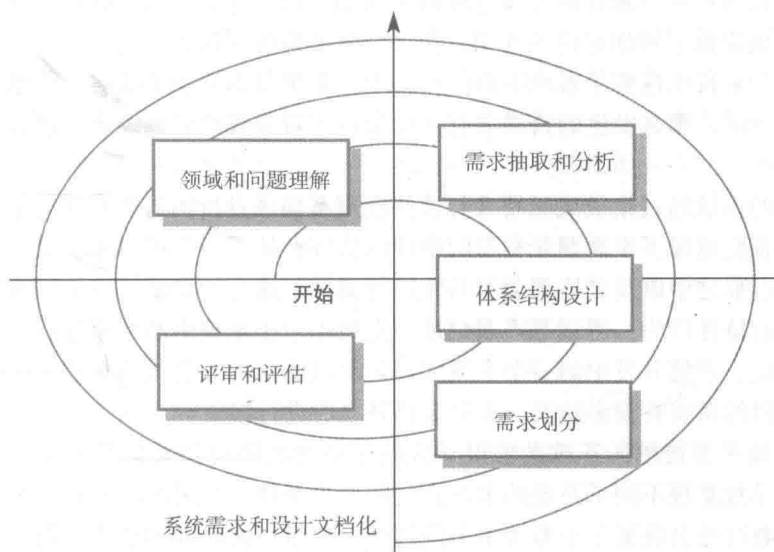


图 19-11 需求和设计的螺旋结构

螺旋结构反映了在现实中需求和设计决策相互影响，因此这些过程相互交错就成为自然

了。从中心位置开始,在现实中螺旋的每一圈都可能会为需求和设计增加新的细节。随着体系结构中的子系统逐渐被识别出来,这些子系统实现系统需求的职责也逐渐得到确定。螺旋中有些圈可能关注需求,其他一些则关注设计。有时需求和设计过程中收集的新知识意味着问题陈述自身也必须修改。

对于几乎所有的系统而言,实现需求都有很多种可能的设计。这些设计覆盖了硬件、软件和人的操作的很多不同的组合。你所选择的用于进一步开发的方案可能是满足需求的最合适的技术方案。然而,在更广阔范围内的组织和政治上的考虑可能会影响方案的选择。例如,一个政府客户可能会倾向于在系统中使用国内而不是国外的供应商,即使国内产品在技术上差一些。

这些影响通常会在螺旋模型的评审和评估阶段产生,在此阶段中设计和需求可能会被接受或否决。当评审确定需求和高层设计都已足够详细使得子系统都能进行规格说明和设计了,整个过程便结束了。

子系统工程涉及系统硬件和软件构件的设计和构造。对于某些类型的系统,例如宇宙飞船,所有的硬件和软件构件都可以在开发过程中进行设计和构造。然而,在大多数系统中,一些构件是通过购买而非开发来获得的。购买已有的产品通常比开发特殊目的的构件要便宜。然而,如果购买大型的成品系统,例如 ERP 系统,那么为了让它们在运行环境中正常使用而进行的配置工作需要很高的成本。

子系统通常都是并行开发的。当遇到跨越子系统边界的问题时,将会产生一个系统修改请求。对于包括大量硬件工程的系统而言,开始制造之后进行修改通常都非常昂贵的。经常需要寻找对问题进行补偿的“变通方案”。这些变通方案通常都需要修改软件以实现新需求。

在系统集成过程中,将独立开发的子系统放到一起来搭建一个完整系统。这一集成可以通过“大爆炸”式的方法来实现,此时所有的子系统在同一时间被集成到一起。然而,出于技术和管理上的原因,一次集成一个子系统的增量集成过程是最好的办法。

1. 通常无法对所有子系统的开发进度做出规划,使得它们在同一时间开发完成。

2. 增量集成降低了错误定位的开销。如果许多子系统同时集成到一起,那么测试过程中发现的错误可能来自于这些子系统中的一个。如果将单个子系统与一个已经可以工作的系统进行集成的话,那么发生的错误很有可能来自于新集成的子系统或者已有子系统与新子系统之间的交互。

越来越多的系统通过集成成品硬件和软件应用系统来进行构造,实现与集成之间的区分越来越模糊。有些情况下不需要开发新的硬件或软件。从根本上讲,系统集成是系统的实现阶段。在集成过程之中以及之后都会对系统进行测试。这时的测试应该关注测试构件之间的接口以及系统的整体行为。测试不可避免地会发现单个子系统中必须要进行修复的问题。测试需要很长时间,系统开发中的一个常见问题是测试团队可能会耗尽预算或时间。这个问题可能导致所交付的系统有很多错误,需要在部署之后进行修复。

由于对其他子系统的不正确的假设导致的子系统故障经常会在系统集成过程中暴露出来。这可能会导致实现不同子系统的承包商之间发生争执。如果问题是在子系统交互中发现的,那么承包商可能会就哪个子系统出了问题进行争辩。协商如何解决问题可能需要花费数周或数月。

系统开发过程的最终阶段是系统交付和部署。软件在硬件上进行安装并准备运行。这可能包括更多的系统配置以反映系统使用时所处的本地环境、从现有系统中进行数据转换,以

及准备用户文档和培训。在此阶段,你可能还必须对环境中的其他系统进行重新配置以保证新系统能够与它们实现互操作。

虽然系统部署原则上很直观,但是经常比所预想的要难。用户环境可能与系统开发者所预想的不一樣。对系统进行适应性调整以使其能够在一个非预期的环境下工作可能会很难。已有的系统数据可能需要进行大量的清洗,其中一些可能会比预想的需要更多的工作量。与其他系统的接口可能没有进行适当的文档描述。你可能会发现所规划的运行过程因为与其他系统的运行过程不兼容而必须改变。用户培训经常很难安排,从而导致用户在开始使用系统后无法充分使用系统的能力。系统部署可能会因此比预期花费更长的时间和更多的成本。

19.5 系统运行和演化

运行过程是指按照系统设计者所设想的方式使用系统时所包含的过程。例如,一个空中交通控制系统的操作人员在飞机进入和离开空域、飞机必须改变高度或速度、发生紧急状况等情况下遵循特定的过程。对于新系统而言,这些运行过程必须在系统开发过程中进行定义和描述。操作人员必须进行培训,其他工作过程必须进行适应性调整以使新系统能够得到有效使用。未发现的问题可能会在这个阶段被发现,因为系统规格说明可能包含错误或遗漏。即使系统运行与规格说明一致,系统的功能可能并不满足真正的运行需要。结果,操作人员可能无法按照设计者所设想的那样使用系统。

虽然运行过程的设计者可能已经是在广泛的用户调研基础上进行的过程设计,但是在用户适应新系统并且探索如何使用系统的实践过程中总是有一段“驯化”期(Stewart and Williams 2005)。虽然用户界面设计很重要,但是一些研究表明随着时间推移用户可以逐渐适应复杂的界面。随着用户经验的不断积累,他们更倾向于以一种快捷的方式使用系统而不是追求容易使用。这意味着设计系统时不应该只是迎合缺少经验的用户,而应该让用户界面能够适应有经验的用户。

一些人认为系统操作人员是系统问题的根源之一,因此应该转向能够将人的参与最小化的自动化系统。这种认识有两个问题。

1. 这很有可能会增加系统的技术复杂性,因为系统设计必须应对所有可以预见的失效模式。这增加了构造系统所需的成本和时间。此外还需要考虑引入人来处理非预期的失效。
2. 人具有很强的适应性,可以处理各种问题以及非预期的情形。因此,当进行系统规格说明和设计时,不需要预计所有可能出错的地方。

人具有独特的能力可以有效应对非预期的情形,即使他们对于这些非预期的事件或系统状态并没有直接经验。因此,当出现问题的时候,系统操作人员经常可以通过找到变通的方法以及以非标准的方式使用系统来实现问题恢复。操作人员还可以使用他们的本地知识来调整和改进过程。通常,实际的运行过程与系统设计者所设想的过程并不相同。

因此,应该将运行过程设计得灵活并具有适应性。运行过程的限制不应该太多,不应该要求按照固定的顺序完成相关操作,而系统软件不应该依赖于所遵循的特定过程。操作人员通常会根据自己所知的在现实情况下什么可行什么不可行来对过程进行改进。

当系统投入运行后可能出现的一个问题是新系统与老系统一起运行。这可能导致硬性的不兼容问题,或者难以在系统之间进行数据传输。其他更加微妙的问题也可能出现,因为不同的系统的用户界面不同。引入一个新系统可能导致操作人员的差错率提高,因为操作人员可能会将用户界面命令用到错误的系统上。

19.5.1 系统演化

大型复杂系统通常生存周期都很长。复杂的硬件 / 软件系统可以持续使用超过 20 年，即使最初的硬件和软件技术已经被淘汰了。这主要是因为几个方面的原因，如图 19-12 所示。

因 素	原 理
投资成本	一个系统工程项目的成本可能高达几千万甚至几亿美元。只有系统能够持续多年为组织提供价值才能证明这些成本的正当性
专业知识的丢失	随着业务变化和重组以关注核心活动，他们经常失去工程专业知识。这可能意味着他们缺少为一个新系统制定需求规格说明的能力
更换成本	更换一个大型系统的成本很高。更换已有系统只有当更换后与已有系统相比可以显著节约成本才会具有说服力
投资回报	如果系统工程的总体预算是固定的，那么将预算花在其他业务领域的新系统上的投资回报可能会比更换已有系统要高
变更的风险	系统是业务运行的固有部分，用新系统替换已有系统的风险很难让人接受。新系统的危险是其中的硬件、软件和运行过程都有可能出问题。这些问题对于业务造成影响的潜在损失可能会很高，以至于人们不愿意承受系统更换的风险
系统依赖	系统之间相互依赖，因此更换其中一个系统可能会导致其他系统的大范围变化

图 19-12 影响系统生存周期的因素

在整个生存期之中，大型复杂系统不断变化和演化以纠正原始系统需求中的错误并且实现新出现的需求。系统中的计算机很可能被替换为新的更快的机器。使用系统的组织可能会进行重组并以不同的方式使用系统。系统的外部环境可能会变化，迫使系统发生变化。因此，演化是一种与正常的系统运行过程相伴随的一个过程。系统演化包括重新进入开发过程来对系统的硬件、软件和运行过程进行修改和扩展。

系统演化与软件演化（在第 9 章中介绍）一样，成本很高，这是因为以下几个原因。

1. 所提出的变更必须非常仔细地从业务和技术两方面进行分析。变更必须有利于系统的目标并且不应该仅仅是由技术驱动的。
2. 因为子系统从来就不是完全独立的，对一个子系统的修改可能会产生副作用，从而对其他子系统的性能或行为带来负面影响。因此，这些子系统后续可能需要进行修改。
3. 最初的设计决策的原因经常没有记录下来。负责系统演化的人不得不自己找出某个设计决策背后的原因。
4. 随着系统使用时间越来越长，系统结构由于变更而逐渐老化，进一步进行变更的成本因此也越来越高。

使用了很多年的系统在过时的硬件和软件技术上经常是比较可靠的。这些“遗留系统”（在第 9 章中介绍）是使用过时的技术开发的基于计算机的社会技术系统。然而，它们不仅仅包含遗留硬件和软件。它们还依赖于遗留的过程和规程——老的做事情的方式，这些方式很难变化，因为它们依赖于遗留软件。对系统某个部分的变动不可避免地会引起对其他构件的变动。

在系统演化过程中对系统的变更经常是问题和漏洞的来源。如果实施变更的人不是开发系统的人,那么他们可能会不了解某个设计决策的目的是为了可依赖性和信息安全。因此,他们可能会变更系统,丢失系统最初开发时有意实现的安全防护。而且,由于测试十分昂贵,不可能对每一个系统变更都进行完整的重新测试。因此,测试可能无法发现这些变更所导致的引发其他构件出错的副作用。

要点

- 系统工程关注与复杂社会技术系统的需求规约、购买、设计和测试相关的所有方面。
- 社会技术系统包括计算机硬件、软件和人,并且位于一个组织之中。这些系统被设计用来支持组织或业务目标和目的。
- 一个系统的涌现特性是该系统作为整体的特性,不是构成系统的构件的特性。这些特性包括性能、可靠性、可用性、安全性和信息安全等。
- 基本的系统工程过程包括概念系统设计、系统采购、系统开发、系统运行。
- 概念系统设计是一个关键活动,在此过程中会开发高层系统需求以及关于系统运行的愿景。
- 系统采购包括确定购买什么系统以及谁来作为系统供应商的所有活动。成品应用系统、可配置的商用成品系统以及定制化开发的系统使用不同的采购过程。
- 系统开发过程包括需求规格说明、设计、构造、集成和测试。
- 系统投入使用后,系统的运行过程以及系统自身不可避免地会发生变化以反映业务需求以及系统环境的变化。

阅读推荐

《Airport 95: Automated Baggage System》是一个精彩、易读的案例研究,包括系统工程项目可能出什么问题以及软件是如何造成更大范围的系统失效的。(ACM Software Engineering Notes, 21, March 1996) <http://doi.acm.org/10.1145/227531.227544>

《Fundamentals of Systems Engineering》是美国国家航空和宇航局系统工程手册中的引言章节,其中给出了空间系统的系统工程过程的概述。虽然这些系统基本都是技术性系统,但也有一些需要考虑的社会技术问题。其中,可依赖性显然极其重要。(NASA Systems Engineering Handbook, NASA-SP 2007-6105, 2007) http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080008301_2008008500.pdf

《The LSCITS Socio-technical Systems Handbook》这本手册以一种容易理解的方式介绍了社会技术系统,并且提供了一些更加详细地介绍社会技术问题的论文的访问方式。(一系列不同的作者, 2012) <http://archive.cs.st-andrews.ac.uk/STSE-Handbook>

《Architecting systems: Concepts, Principles and Practice》是一本与众不同、让人眼前一亮的系统工程著作,不像许多“传统”的系统工程著作那样主要关注硬件。作者是一个很有经验的系统工程师,他从大量的系统中吸收有用的例子,并且认识到了社会技术问题与技术问题一样重要。(H. Sillitto, College Publications, 2014)

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap19/>

支持视频的链接: <http://software-engineering-book.com/videos/systems-engineering/>

练习

- 19.1 给出两个由复杂的社会技术系统支持的政府职能的例子,解释为什么在可预见的未来这些职能无法完全自动化。
- 19.2 为什么基于计算机的系统所安装的环境可能会对系统产生未预期的导致系统失效的影响?
- 19.3 为什么无法从系统构件的属性推测一个复杂系统的涌现特性?
- 19.4 什么是“非常规问题”?解释为什么要把一个国家医疗档案系统的开发视为一个“非常规问题”。
- 19.5 为欧洲博物馆联合会开发一个多媒体虚拟博物馆系统,提供古希腊的虚拟体验。该系统应当通过 Web 浏览器为用户提供观看古希腊 3D 模型的手段,还要支持沉浸式的虚拟现实体验。为这样的系统开发一个概念设计,突出其中的关键特性以及基本的高层需求。
- 19.6 为什么需要保持灵活性,并在采购大型成品软件系统(例如 ERP 系统)时对系统需求进行适应性调整?通过网络搜索查找关于此类系统失效的讨论,并且从社会技术的角度解释为什么会发生这些失效。下面是一个可能的起点: <http://blog.360cloudsolutions.com/blo3g/bid/94028/> (6 个 ERP 系统失效典型案例)。
- 19.7 为什么系统集成是系统开发过程的一个非常重要的部分?列举 3 个可能导致系统集成出现困难的社会技术问题。
- 19.8 为什么遗留系统对于业务运行通常很重要?
- 19.9 针对将系统工程视为与电子工程或软件工程一样的职业有哪些正面和反面的论据?
- 19.10 你是一个参与某财务系统开发的工程师。在安装过程中,你发现该系统将会导致很多人失业。该项目中相关的人拒绝你访问完成系统安装所需的一些重要信息。作为一个系统工程师,你应该在何种程度上置身其中?按照合同完成系统安装是你的职责吗?你应该简单地放弃安装直到采购单位解决问题吗?

参考文献

Baxter, G., and I. Sommerville. 2011. “Socio-Technical Systems: From Design Methods to Systems Engineering.” *Interacting with Computers* 23 (1): 4–17. doi:10.1016/j.intcom.2010.07.003.

Checkland, P. 1981. *Systems Thinking, Systems Practice*. Chichester, UK: John Wiley & Sons.

Fairley, R. E., R. H. Thayer, and P. Bjorke. 1994. “The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications.” In *1st Int. Conf. on Requirements Engineering*, 40–7. Colorado Springs, CO. doi:10.1109/ICRE.1994.292405.

IEEE. 2007. “IEEE Guide for Information Technology. System Definition—Concept of Operations (ConOps) Document.” *Electronics*. Vol. 1998. doi:10.1109/IEEESTD.1998.89424. <http://ieeexplore.ieee.org/servlet/opac?punumber=6166>

Mostashari, A., S. A. McComb, D. M. Kennedy, R. Cloutier, and P. Korfiatis. 2012. “Developing a Stakeholder-Assisted Agile CONOPS Development Process.” *Systems Engineering* 15 (1): 1–13. doi:10.1002/sys.20190.

- Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155-169. doi:10.1007/BF01405730.
- Stevens, R., P. Brook, K. Jackson, and S. Arnold. 1998. *Systems Engineering: Coping with Complexity*. London: Prentice-Hall.
- Stewart, J., and R. Williams. 2005. "The Wrong Trousers? Beyond the Design Fallacy: Social Learning and the User." In *User Involvement in Innovation Processes. Strategies and Limitations from a Socio-Technical Perspective*, edited by H. Rohrache, 39-71. Berlin: Profil-Verlag.
- Thayer, R. H. 2002. "Software System Engineering: A Tutorial." *IEEE Computer* 35 (4): 68-73. doi:10.1109/MC.2002.993773.
- White, S., M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey. 1993. "Systems Engineering of Computer-Based Systems." *IEEE Computer* 26 (11): 54-65. doi:10.1109/ECBS.1994.331687.

系统之系统

目标

本章的目标是介绍系统之系统的概念，并讨论构建复杂的软件系统之系统的挑战。阅读完本章后，你将：

- 理解系统之系统的意义以及系统之系统与独立系统之间的区别；
- 理解系统之系统的分类以及不同类型的系统之系统之间的区别；
- 理解为何基于还原论的传统软件工程方法不适用于系统之系统的开发；
- 了解系统之系统工程过程以及系统之系统的体系结构模式。

我们需要软件工程是因为我们创建了庞大且复杂的软件系统。软件工程的规范出现在 20 世纪 60 年代，因为那时构建大型软件系统的初步尝试多以失败告终。创建软件的花费比预期的要多，耗时比计划的要长，软件本身也常常并不可靠。为了解决这些问题，人们开发了一系列非常成功的软件工程技巧和技术。现在我们可以构建比 20 世纪 70 年代更加庞大、更加复杂、更加可靠也更加有效的软件系统。

然而，我们还未“解决”大型系统工程中的问题，软件项目的失败仍很常见。例如，美国和英国的政府医疗保健系统实现过程中都存在严重的问题以及延期。这些问题的根源在于，相比 20 世纪 60 年代，现在人们创建的系统要更庞大也更复杂。人们在尝试创建这些“超大系统”时使用的方法和技术并不是为此设计的。在本章后面的部分将会讨论，目前的软件工程技术并不能应对许多现在的系统中固有的复杂度。

从软件工程的观念被提出以来，软件系统的大小已经增加了很多。现在的大型系统可能比 20 世纪 60 年代的所谓“大型”系统要大成百上千倍。2006 年，Northrop 及其同僚 (Northrop et al. 2006) 认为我们很快就会看到拥有超过 10 亿行代码的系统。在这一预测过去近 10 年后的今天，这样的系统已经存在。

当然，我们并不是从零开始写出 10 亿行代码。如第 15 章中论述的，软件工程真正的成功经验始终是软件复用。正是由于我们开发了跨应用和跨系统的软件复用方法，大型系统开发才成为可能。现在和未来的超大型系统将会通过整合现有的来自不同供应商的系统以创建系统之系统来构建。

何谓系统之系统 (Systems of Systems, SoS)？正如 Hitchens 所说 (Hitchens 2009)，从一般系统的角度来讲，系统与系统之系统间并无任何区别，它们都有涌现特性，可以由多个子系统构成。然而，从软件工程的角度来讲，笔者认为两者间有一个区别，这个区别更多是社会技术上的而非技术上的，认识到这一区别是十分有意义的。

系统之系统是包含两个或更多独立管理的单元的系统。

这意味着系统之系统中并不存在一个对系统各个部分进行管理的管理者，系统的不同部分分别对应不同的管理和控制的策略和规则。正如我们将看到的，分布式的管理和控制对系统总体的复杂度有深远的影响。

上面对系统之系统的定义并没有提到系统之系统的大小。包含不同提供商提供的服务的较小的系统也是系统之系统。一些系统之系统工程中的问题对这种小型系统也适用,但如果系统的组成部分本身是大型系统时,就会遇到真正的挑战。

系统之系统领域的大多数工作来源于国防工业。20世纪末,随着软件系统容量的增加,整合并控制原先独立的军事系统(如海军和陆基的防空和反舰系统)成为可能。这种系统可能包含几十个或几百个独立的单元,软件系统负责跟踪记录这些单元并向控制器提供信息,使它们可以以最有效的方式部署。

这类系统之系统已经超出了软件工程书籍的范畴。相反,本书只关注一类系统之系统,其系统单元是软件系统而非飞机、军用车辆或雷达之类的硬件。系统之系统通过整合独立的软件系统来创建,在本书写作时,多数软件系统之系统只包含少量的独立系统,系统之系统的每个组成部分通常是一个独立的复杂系统。然而,有预测称,在未来几年内,随着越来越多的软件被整合以利用它们提供的功能,软件系统之系统的大小可能显著增长。

软件系统的系统之系统的样例有:

1. 云管理系统,负责处理本地私有云管理和服务器公有云(如亚马逊和微软)管理;
2. 在线银行系统,负责处理借贷请求并与信用调查机构提供的征信系统相连以检查申请人信用状况。
3. 应急信息系统,整合来自警方、救护机构、火警和海岸警卫队的处理民间紧急状况(如洪水和大规模事故)的可用资源信息。

4. 本书第1章介绍的数字学习环境(iLearn)。该系统通过整合多个独立软件系统,如微软 Office 365、虚拟学习环境(如 Moddle)、模拟模型工具和新闻档案之类的内容,来提供一系列学习支持。

Maier (Maier 1998) 指出了系统之系统的5个基本特征。

1. 单元的操作独立性。系统的各个部分不仅是系统构件,也可以独立地作为有用的系统运转。系统之系统内的各个子系统独立发展。

2. 单元的管理独立性。系统的各个部分是由不同的组织或一个更大组织的不同部分所“拥有”和管理的。因此,这些系统的管理和演化会使用不同的规则和策略。正如之前提到的,这是区别系统之系统与系统的关键因素。

3. 演化式开发。系统之系统不是在某一个项目中开发的,而是随着时间从组成它们的系统演化而来的。

4. 涌现。系统之系统具有涌现特性,这些特性只有在系统之系统被创建后才会出现。当然,如第19章所述,涌现是所有系统的一个特性,但这一特性在系统之系统中尤为重要。

5. 单元的地理分布。系统之系统的单元经常在地理上分布于不同的组织。这在技术上十分重要,因为这意味着一个在外部管理的网络是系统之系统的重要组成部分。从管理上讲,这一点也很重要,因为这增加了相关的系统在做出系统管理决定时相互交流的难度,并增加了维持系统安全的难度。^①

本节在 Maier 提出的列表的基础上增加以下两点与软件系统之系统特别相关的两点特性。

1. 数据密集。软件系统之系统通常依赖并且管理数量庞大的数据,这些数据在大小上可

① Maier, M. W. 1998. "Architecting Principles for Systems-of-Systems." *Systems Engineering* 1 (4): 267-284. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

能比系统组成本身的代码要大数十甚至数百倍。

2. 异构性。软件系统之系统中的不同子系统未必是用相同的编程语言和设计方法开发的,这是软件技术快速演化所导致的。当开发方法和工具出现新的、完善的版本时,软件公司也经常进行更新。在一个大型系统之系统 20 年的生命周期中,使用的技术可能改变 4 次或 5 次。

正如 20.1 节中所述,这些特性意味着系统之系统会比那些只有单一所有者和管理者的系统复杂得多,我们无法用目前的软件工程方法和技术来应对这样的复杂度。因此,我们目前开发的超大型和非常复杂的软件系统无可避免地会出现问题。我们需要一套全新的软件系统之系统工程的抽象、方法和技术。

许多作者都认识到了这一需求。在英国,一份发表于 2004 年的报告(Royal Academy of Engineering 2004)导致了一次对大型复杂 IT 系统的国家级研究和培训行动(Sommerville et al. 2012)。在美国,软件工程研究所在 2006 年发布了对超大型系统的报告(Northrop et al. 2006)。在系统工程界,Stevens (Stevens 2010)论述了构建交通、医疗和国防领域中的“大型系统”时的问题。

20.1 系统复杂度

在引言中曾提到,构建软件系统之系统时出现工程问题的根源是这些系统固有的复杂度。在本节中,将解释系统复杂度的基本原理,并论述软件系统之系统中出现的不同类型的复杂度。

所有系统都是由互相关联的部分(单元)组成的。例如,一个程序的部分可能是对象,每一个对象的部分可能是常量、变量和方法。关联关系的例子包括“调用”(方法 A 调用方法 B)、“继承自”(对象 X 继承对象 Y 的方法和属性)和“从属于”(方法 A 从属于对象 X)。

所有系统的复杂度都取决于系统单元之间关联关系的数量和类型。图 20-1 展示了两个系统的例子。系统(a)是一个相对简单的系统,其单元间只有少量关联关系。与之相对的是系统(b),其单元数量虽然与系统(a)一样,但由于它具有更多的单元间关联关系,其复杂度也更高。

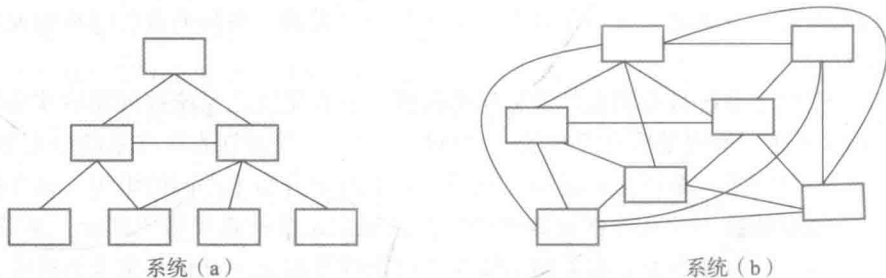


图 20-1 简单系统与复杂系统

关联关系的类型也会影响系统整体的复杂度。静态关联关系是设计好的,可以从系统的静态描述中分析出来。因此,软件系统中“使用”这一关联关系是静态关联关系。不论是通过软件源代码还是通过系统的 UML 模型,都可以分析出任意一个系统构件是如何使用其他构件的。

动态关联关系是系统运行时存在的关联关系。“调用”是动态关联关系，因为在所有含有 if 语句的系统中，你无法确定一个方法是否会调用另一个方法，这取决于系统运行时的输入。动态关联关系更难以分析，因为除了需要系统源代码之外，还需要知道系统输入和使用的数据。

除了系统复杂度，我们还需要考虑系统的开发过程和系统投入使用后的维护过程的复杂度。图 20-2 描绘了这些过程以及它们与已开发的系统之间的关系。

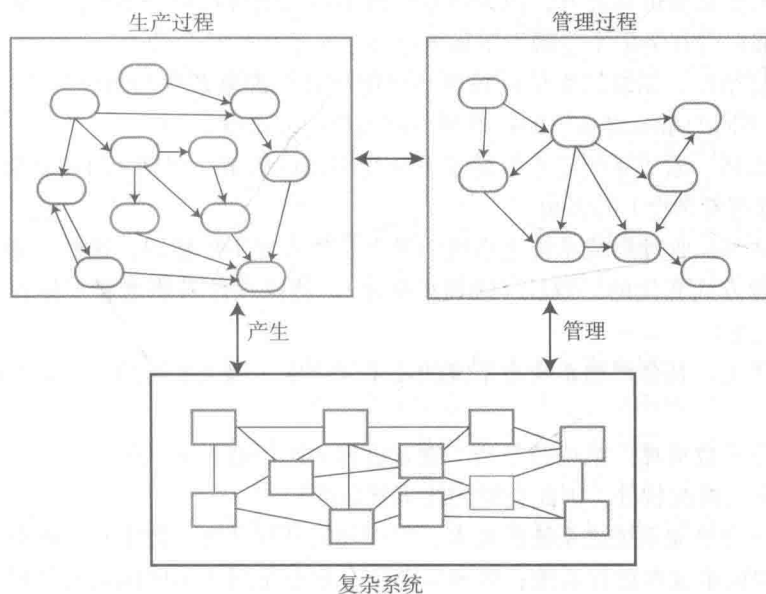


图 20-2 生产和管理过程

随着系统大小增加，系统需要的生产和管理过程也更加复杂。复杂过程本身也是复杂系统，它们难以理解，并且可能有一些意想不到的涌现特性。复杂过程比简单过程耗时更长，在系统开发时需要更多的文档以及相关人员和组织间的协作。生产过程的复杂度是项目出问题的主要原因之一，它会导致项目无法及时交付或是超出预算。因此，大型系统始终存在开销和时间超出限度的风险。

复杂度对软件工程至关重要，因为它是影响系统可读性和可变性的主要因素。系统越复杂，理解和分析系统也就越困难。由于复杂度与系统中单元之间关联关系的数量呈正相关，大型系统也就无可避免地要比小型系统更复杂。随着系统复杂度的提升，系统中单元之间关联关系的数量也更多，对系统中某个部分的改变导致对系统其他某处产生意想不到的影响的可能性也会增加。

与社会技术系统相关的几种不同类型的复杂度包括：

1. 系统的技术复杂度源于系统自身的不同构件之间的关联关系。
2. 系统的管理复杂度源于系统与其管理者之间的关联关系（也就是说，管理者能改变系统中的什么东西）和系统各部分的管理者之间的关联关系。
3. 系统的治理复杂度取决于影响系统的法律、规定和政策之间的关联关系以及为系统负责的组织内部的决策过程之间的关联关系。由于系统的不同部分可能属于不同国家和不同组织，系统之系统内的各个系统可能应用不同的法律、规定和政策。

管理复杂度和治理复杂度有一定相关性，但并不完全一致。管理复杂度是一个操作问题——对系统可以做什么，不可以做什么。治理复杂度与影响系统的组织中更高层的决策过程有关，这些决策过程受国家和国际法律法规约束。

例如，一个公司决定允许其员工使用他们自己的移动设备访问其系统，而不必使用公司提供的笔记本电脑。这个决定本身是一个治理决定，因为它改变了公司的政策。这一决定将导致系统的管理更加复杂，因为管理者需要保证移动设备配置正确，以保护公司数据的安全。系统的技术复杂度也将提升，因为该系统将不再只实现在单个平台上，而可能需要修改软件以使其能够运行在笔记本电脑、平板电脑和手机上。

除了技术复杂度，系统之系统的特性还可能导致管理复杂度和治理复杂度的大幅提升。图 20-3 概括了不同的系统之系统的特性对不同类型的复杂度的影响。

- 1. 操作独立性。组成系统之系统的子系统受不同的政策、规则（治理复杂度）以及管理系统的方式（管理复杂度）的影响。
- 2. 管理独立性。管理组成系统之系统的子系统的方式不尽相同，这些管理方式需要相互协作来保证管理方式变化的一致性（管理复杂度）。特殊软件可能需要支持管理和演化的一致性（技术复杂度）。
- 3. 演化式开发。其会增加系统之系统的技术复杂度，因为系统的不同部分可能使用不同技术开发。
- 4. 复杂度会导致涌现。系统越复杂，它就更容易产生意想不到的涌现特性。抵消这些涌现特性需要开发或修改软件，因此会增加技术复杂度。
- 5. 地理分布会增加系统之系统的技术、管理和治理复杂度。技术复杂度会增加是因为需要软件来协调和同步这些远程系统；管理复杂度会增加是因为不同国家的管理者要协调行动会变得更加困难；治理复杂度会增加是因为系统的不同部分可能位于不同的行政辖区，因此受不同法律和规定的制约。
- 6. 由于数据项之间的关联关系，数据密集型系统技术上也更复杂。数据的错误和不完整也可能增加技术复杂度。由于治理数据使用的法律法规不同，治理复杂度也可能增加。
- 7. 系统异质性会导致技术复杂度增加，因为需要保证系统不同部分使用的不同技术之间的兼容性。

系统之系统的特性	技术复杂度	管理复杂度	治理复杂度
操作独立性		×	×
管理独立性	×	×	
演化式开发	×		
涌现	×		
地理分布	×	×	×
数据密集	×		×
异质性	×		

图 20-3 系统之系统的特性与系统复杂度

今天的大型的系统之系统的复杂度不可思议，无法作为一个整体被理解或分析。正如 20.3 节中将论述的，系统中各部分之间大量的交互以及这些交互天然的动态性，导致传统的工程方法并不适用于复杂系统。开发大型软件密集型系统的项目时出现的种种问题的根源是复杂度，而非管理不善或技术不足。

20.2 系统之系统的分类

前文曾提到,系统之系统的突出特点是它具有两个或两个以上独立管理的单元。不同的具有不同优先权的人都可以做出改变系统日常操作的决策,由于这些人的工作可能并不一致,导致的冲突就需要大量的时间和精力来解决。因此,系统之系统总是具有一定程度的管理复杂度。

然而,上述广义的系统之系统的定义覆盖了很多不同的系统类型,包括那些由某一个组织所拥有但由该组织不同部分管理的系统,以及由不同组织所拥有和管理的系统,并且在某些时候这些组织可能互相竞争。Maier (Maier 1998) 设计了基于其治理复杂度和管理复杂度的系统之系统分类表。

1. 受控系统。受控系统之系统由单个组织所拥有,通过整合该组织所拥有的系统开发。系统单元可能由组织的多个部分独立管理,但组织中存在一个最高管理部门,可以为系统管理设定优先级。该部门可以解决系统不同单元的管理者之间的冲突,因此受控系统具有一定的管理复杂度,但没有治理复杂度。受控系统之系统的一个例子是军事指挥控制系统,该系统可以整合来自空中和地面系统的信息。

2. 协作系统。协作系统之系统是没有负责设定管理优先级或解决冲突的中央机构的系统。通常,这类系统的单元由不同组织所拥有和管理,但这些组织可以通过对系统的共同管理互惠互利。因此,它们通常会设立一个志愿治理体来对系统进行决策。协作系统具有管理复杂度以及有限的治理复杂度。协作系统之系统的一个例子是公交信息整合系统,公共汽车、轨道交通和航空运输提供商一起将各自的系统连接起来,以将最新的信息提供给乘客。

3. 虚拟系统。虚拟系统中不存在中央式管理,系统的参与者可能对系统的总体目的并未达成一致,它们可以选择加入或离开系统之系统。系统的互操作性因取决于可变的公共接口而无法保证。这类系统同时具有非常高的管理和治理复杂度。虚拟系统之系统的一个例子是自动高速算法交易系统,来自不同公司的交易系统互相购买和售出股票,这些交易发生在几分之一秒内。

不幸的是,笔者认为 Maier 采用的命名并没有真正反映这些不同类型系统间的差异。正如 Maier 自己所说,对系统单元的管理总是会涉及协作,因此“协作系统”并不是一个很好的命名。“受控系统”这一名称隐含着自上而下的治理权,但即使是在单个组织中,为了维持相关人员良好工作关系,管理通常是达成一致而非强迫。

“虚拟”系统之系统没有正式的协作机制,但系统参与者可以在系统中互惠互利。因此,参与者可能以非正式的方式进行协作,来保证系统可以持续运转。此外, Maier 所用的“虚拟”一词可能令人困惑,因为现在“虚拟”的意思是“通过软件实现”,比如虚拟机和虚拟现实。

图 20-4 描绘了不同类型系统中的协作。与 Maier 不同,这里所使用的名称更具描述性:

1. 组织型系统之系统的治理层和管理层处在同一个组织或公司中。这类系统之系统对应 Maier 所说的“受控系统之系统”,系统所有者之间的协作由组织管理。这种系统之系统在地理上可能是分布式的,系统的不同部分受不同国家的法律法规管制。在图 20-4 中,系统 1、2 和 3 独立管理,但系统的治理是集中(中央)式的。

2. 联邦型系统的治理依赖于一个由所有系统所有者的代表组成的自愿参与体。图 20-4

中，系统 1、2 和 3 参与组成了一个治理体。系统所有者们同意协作并相信治理体所做出的决定具有约束力。他们根据各自的管理策略实现这些决定，尽管由于国家法律、法规和文化的差异，其实现可能不同。

3. 联合型系统没有正式的治理机制，但相关组织以非正式的方式进行协作并管理各自的系统，使系统保持为一个整体。例如，如果某个系统给其他系统提供某个数据，该系统管理者不会在不通知其他系统的情况下就改变数据格式。

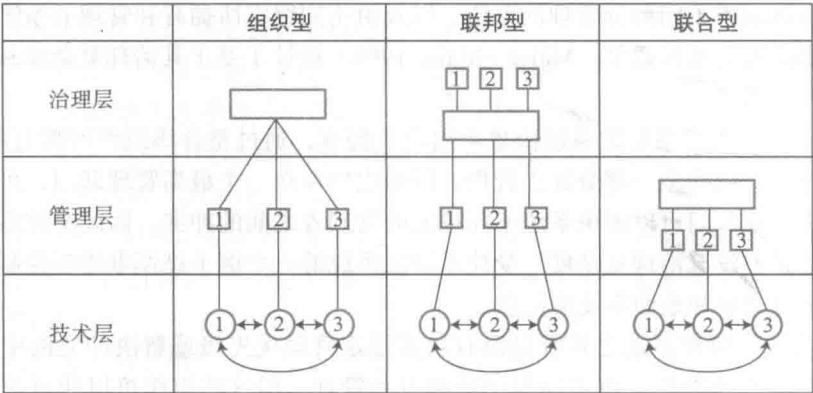


图 20-4 系统之系统的协作

基于治理的分类表为我们提供了识别一个系统之系统的治理需求的手段。通过根据该模型划分一个系统的类型，你可以检验是否存在合适的治理结构，以及这些结构是否是你真正需要的。在组织间建立这些组织结构是一个政治过程，而且会无可避免地花费很长时间，因此在过程的早期理解治理问题并采取行动来保证合适的治理的方式是非常有用的。有时可能需要采用将一个系统从某个分类移动到另一分类的治理模型。在图 20-4 中，向左移动治理模型通常可以减少复杂度。

前文中曾提到校内数字学习环境（iLearn）是一个系统之系统。它不仅与数字学习系统相连，还和学校行政系统及网络管理系统相连。这些网络管理系统的作用是互联网过滤，即防止学生访问互联网上不该他们访问的内容。

iLearn 在技术上比较简单，但由于教育的赞助方和管理方式，系统的治理复杂度很高。在许多国家，大学前教育的赞助和组织是地方级的而非国家级的，也就是说，州、城市或是郡对当地的学校负责，并具有决定学习资金和政策的自治权。每个地方政府都有自己的行政系统和网络管理系统。

苏格兰有 32 个地方政府机构对当地的教育负责，学校行政系统外包给三个提供商之一，iLearn 必须接入这些系统。但是每个地方政府机构有各自的网络管理政策，涉及不同的网络管理系统。

开发数字学习系统是国家计划，但要建立数字学习环境，系统需要与网络管理和学校行政系统整合。因此，数字学习系统是一个包含行政和网络管理系统以及其他 iLearn 内的系统（如 Office365 和 Wordpress）的系统之系统。多个机构之间没有共同的治理过程，因此，根据分类表，这是一个系统的联合。实际上，这意味着由于不同的互联网过滤政策，它无法保证不同地区的学生能够获取相同的工具和内容。

在生成了系统的概念模型后,我们强烈建议地方政府机构间应出台一个共同的行政信息供应和互联网过滤政策。事实上,我们建议该系统应该是一个联邦型系统而非联合型系统。这一建议需要建立一个新的治理体系来为系统达成一个共同的政策和标准。

20.3 还原论和复杂系统

前文已提到,我们现在的软件工程方法和技术已经不能处理其从现代系统之系统继承的复杂性。当然,这并不是个新观点:所有工程学科中的进步都是被挑战和疑难问题所驱动,新的方法和工具也都是为了应对现存方法中的失效和困难而被开发的。

在软件工程中,我们已经目睹了这个学科在帮助管理软件系统日益增长的规模和复杂性方面不可思议的快速发展,这确实是有价值的努力。我们现在能构建比 20 世纪六七十年代在数量级上更大、更复杂的系统。

与其他工程类学科一样,软件工程中复杂管理的基础方法被称为还原论(reductionism)。还原论是基于任何系统都是由部分和子系统组成这样一个哲学立场,它假设系统整体的行为和性质能够通过理解个体部分和这些部分之间的关系被理解和预测。因此,在设计一个系统时,组成这个系统的部分应该被确定好和分别构建,接着被装配进整个系统。系统可以看作是有层级的,同时在层级中有重要的父节点和子节点的关系。

还原论已经并且将继续是所有工程类学科中的基础方法。我们可以确定系统中相同类型部分的一般抽象,并且分别设计和构造这些抽象,它们能够被整合起来创造所需系统。例如,汽车中的抽象可以是车身外壳、传动系统、发动机、燃油系统等。这些抽象间的相对关系较少,因此指定接口并且独立地设计和构建系统中的每个部分是可能的。

相同的还原论方法已经成为软件工程的基础,这已长达 50 年之久。在自顶向下设计中,从一个系统的高层模型开始,把它分解成它的构件,这就是一个还原论方法。这是所有软件设计方法的基础,例如面向对象设计。编程语言包括抽象,例如直接地反映还原论系统分解的过程和对象。

敏捷方法,虽然它们看起来可能与自顶向下方法系统设计区别很大,但也是还原论方法。它们的做法是把系统分解成几个部分,分别实现这些部分,接着把这些部分集成起来创建系统。敏捷方法和自顶向下设计方法间唯一的不同是,系统被递增地分解成构件而不是立刻分解。

还原论方法在系统构件间关系或交互作用相对较少的情况下最容易成功,因为这种情况下科学地对这些相互关系建模是可能的。在机械和电气系统中,系统构件间存在物理联系,通常都是符合前述情况的。但是在电子系统和与前者明显不同的软件系统中,系统构件间可能存在更多的静态和动态关系,通常是不符合前述情况的。

人们在 20 世纪 70 年代意识到了软件和硬件构件间的区别。设计方法强调限制和控制系统各部分之间关系的重要性,这些方法建议构件应该紧密集成,但构件间的关系应该是松耦合的。紧密集成意味着构件内部的关系占绝大多数,松耦合意味着构件-构件关系组合相对应该很少。对紧密集成(数据和操作)和松耦合的需求是面向对象软件工程发展的驱动力。

不幸的是,在大型系统中,特别是在系统之系统中,对关系数量和类型的控制几乎是不可能的。当系统中有许多关系或这些关系很难理解和分析时,还原论是不合适的。因此任何类型的大型系统开发都有可能陷入困境。

这些潜在困难的原因是还原论固有的基本假设在大型复杂系统中是不适用的 (Sommerville et al. 2012)。这些假设展示在图 20-5 中, 而且被应用在以下领域。



图 20-5 还原论的假设与复杂系统的现实

1. 系统所有权和控制权。还原论假设系统中存在控制权, 此控制权能解决争论并且做出适用于整个系统的高层技术决策。正如我们所看到的, 因为系统之系统的管理涉及多个机构, 上述假设在系统之系统中是不正确的。

2. 做出理性决策。还原论假设构件间的交互能够通过诸如数学建模的方式进行客观评估, 这些评估是系统决策的驱动力。因此, 如果一个车辆的详细设计说它提供了最佳的燃油经济性并且不会造成功率的降低, 还原论方法就会假设这将会是设计上的选择。

3. 定义系统边界。还原论假设系统边界的存在是能够被承认而且被定义的。这通常是明确的: 存在一个物理外壳对系统做出定义, 诸如, 汽车中的系统, 桥必须跨过给定伸展的水等。复杂系统通常被开发来处理棘手问题 (Rittel and Webber 1973), 对这些问题, 决定什么是系统内部什么是系统外部通常要靠主观判断, 而所涉及的利益相关者的意见通常是不一致的。

这些还原论假设为所有复杂系统做了分解, 但是当这些系统是软件密集型时, 困难进一步加剧了, 具体如下。

1. 软件系统中的关系不受物理定律的约束。我们无法创建能预测软件系统行为和属性的数学模型。因此, 我们没有决策的科学依据。政治因素通常是大型和复杂软件系统决策的驱动力。

2. 软件没有物理限制; 因此没有有关系统的边界应该在哪里的限制。不同的利益相关者将争论把边界以最有利于他们的方式放置。此外, 相比硬件需求的变更, 软件需求的变更更容易发生。边界和系统的范围在其开发过程中可能会发生变化。

3. 连接来自不同所有者的软件系统相对容易; 因此我们更可能在没有单一管理机构的情况下尝试和创建系统之系统, 并且不能完全控制所涉及的不同系统的管理和演化。

由于这些原因, 笔者认为在大型软件系统工程中, 普遍的问题和困难是不可避免的。政府的大项目的失败, 如英国和美国的卫生自动化项目的失败是一个复杂性的后果, 而不是技术或项目管理的失败。

还原论方法, 如面向对象开发, 在改进我们设计多种类型的软件系统能力方面已经非常成功。这些方法会继续在中小型系统开发方面发挥效用, 它们的复杂性可以被控制并且它

们可能是某个软件 SoS 的一部分。然而，由于还原论的基本假设，“改进”这些方法不会导致我们设计复杂系统之系统能力的提高。相反，我们需要新的抽象、方法和工具来识别 SoS 工程的技术、人力、社会和政治复杂性。相信这些新方法将是概率和统计，工具将依靠系统模拟来支持决策。开发这些新方法是 21 世纪软件和系统工程的一个主要挑战。

20.4 系统之系统工程

系统之系统工程是整合现有系统来创建新功能和能力的过程。系统之系统不是按自顶向下方法设计的，相反，它们是在组织认识到可以将附加价值集成到 SoS 中，在现有系统基础上增加这些价值时创建的。例如，一个城市政府可能希望减少城市特定热点的空气污染。为此，它可能将其交通管理系统与国家实时污染监测系统相结合。然后允许交通管理系统通过改变策略来减少污染，如改变交通灯序列，速度限制等。

软件 SoS 工程问题与第 15 章讨论过的大规模应用系统集成问题有很多共同点 (Boehm and Abts 1999)。简而言之，这些问题是：

1. 缺乏对系统功能和性能的控制；
2. 不同系统的开发人员做出的不同和不兼容的假设；
3. 不同系统有不同的进化策略和时间表；
4. 当出现问题时，缺乏系统所有者的支持。

在构建软件系统之系统中的许多努力来自于定位这些问题。它涉及决定系统架构、开发软件接口来协调参与系统之间的差异，以及使系统适应可能发生的不可预见的变化。

软件系统之系统是大而复杂的实体，并且它们的开发过程变化很大，这取决于开发 SoS 时所涉及系统中通常的类型、应用领域，以及参与开发的组织的需求。如图 20-6 所示，在 SoS 开发过程包含下面 5 个一般意义上的活动。

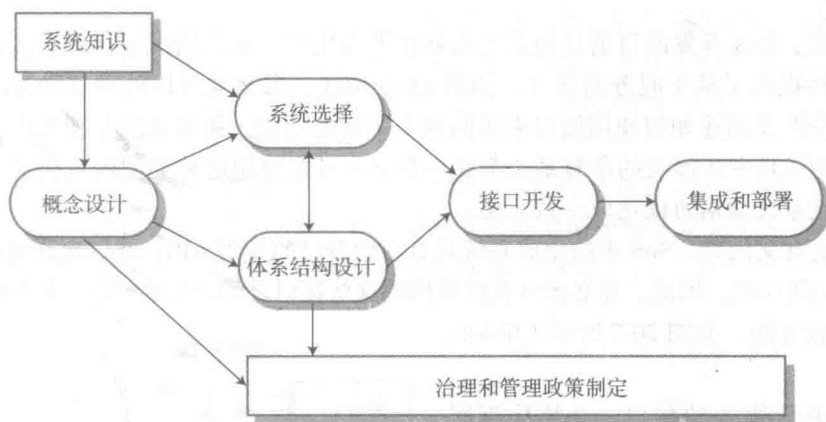


图 20-6 系统之系统工程过程

1. 概念设计。本书在第 19 章中介绍了概念设计的理念，它涵盖了系统工程。概念设计是为系统创造一个高层次的活动，对整个系统定义基本需求和识别约束。在 SoS 工程中，概念设计过程的一个重要输入是可能参与 SoS 的现有系统的知识。

2. 系统选择。在此活动期间，选择包括在 SoS 中的一组系统。这个过程与选择应用系统进行复用的过程类似，见第 15 章。你需要评估现有系统来选择你需要的功能。在你选择

应用系统时，选择标准主要是商业的；也就是说，哪些系统能以你准备支付的价格提供最合适的功能？

然而，政治要求以及系统治理和管理问题通常是影响 SoS 中包含哪些系统的关键因素。例如，一些系统可能被排除在考虑之外，因为一个组织不希望与竞争对手合作。在其他情况下，对系统的联合做出贡献的组织可能会提出某些系统，并且坚持使用它们，即使它们不一定是最好的系统。

3. 体系结构设计。与系统选择同时进行，必须开发 SoS 的整体体系结构。体系结构设计本身就是一个重要的课题，见 20.5 节。

4. 接口开发。SoS 中涉及的不同系统通常有不兼容的接口。因此，开发 SoS 的软件工程工作的主要部分是开发接口，使组成系统可以互操作。这可能还涉及统一用户界面的开发，使 SoS 的操作者在使用 SoS 中的不同系统时不必处理多个用户界面

5. 集成和部署。此阶段使 SoS 中涉及的不同系统通过开发的接口共同工作和互操作。系统部署意味着将系统置于相关组织中并使其可操作。

在进行这些技术活动的同时，需要有一个高级别活动，涉及建立系统之系统的治理政策和确定实施这些政策的管理准则。在涉及若干组织的情况下，这个过程可能是长期和困难的。它可能涉及组织改变自己的政策和流程。因此，重要的是在 SoS 开发过程的早期阶段开始治理讨论。

20.4.1 接口开发

SoS 中的组成系统通常是为了某些特定目的而独立开发的。它们的用户界面是根据原来的目的定制的。这些系统可能具有或可能不具有允许其他系统直接与它们交互的应用编程接口 (API)。因此，当这些系统集成到 SoS 中时，必须开发软件接口，以便允许 SoS 中的组成系统互操作。

一般来说，SoS 开发的目的是使系统能够在没有用户干预的情况下彼此直接通信。如果这些系统已经提供了基于服务的接口，如第 18 章所述，那么就可以使用这种方法实现该通信。接口开发涉及描述如何使用接口来访问每个系统的功能。所涉及的系统可以彼此直接通信。系统联盟（其中所涉及的所有系统都是同位体）可能使用这种类型的直接交互，因为它不需要预先就系统通信协议达成一致意见。

然而，更常见的是，SoS 中的组成系统具有它们自己的专用 API，或仅允许通过其用户接口来访问它们的功能。因此，你必须开发能够协调这些接口之间差异的软件。最好将这些接口实现为基于服务的，如图 20-7 所示（Sillitto 2010）。

要开发基于服务的接口，必须检查现有系统的功能，并定义一组服务以反映该功能。然后接口提供这些服务。服务通过调用底层系统 API 或模仿用户与系统的交互来实现。SoS 中的系统之一通常是管理组成系统之间交互的主体或协调系统。主系统充当服务代理，在 SoS 中的不同系统之间引导服务调用。因此，每个系统不需要知道其他

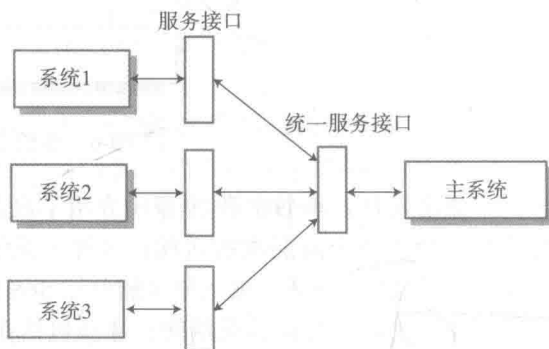


图 20-7 具有服务接口的系统

哪个系统正在提供被调服务。

SoS 中每个系统的用户界面可能不同。主系统必须具有一些总体用户界面来处理用户认证,并提供对底层系统特征的访问。然而,实现统一的用户界面以替换基础系统的各个接口通常是昂贵和耗时的。

统一的用户界面(UI)使新用户更容易学习使用 SoS,并减少用户错误的可能性。然而,统一 UI 开发是否具有成本效益取决于以下几个因素:

1. SoS 中系统的交互假设。一些系统可能具有过程驱动的交互模型,其中系统控制界面并提示用户输入。其他人可以给予用户控制,使得用户选择与系统的交互序列。实际上几乎不可能统一不同的交互模型。

2. SoS 的使用模式。在许多情况下,以这样的方式使用 SoS,使得一个站点上的用户的大多数交互是与某个组成系统完成的。仅当需要其他信息时,它们才使用其他系统。例如,空中交通控制器通常可以使用雷达系统获取飞行信息,只有在需要附加信息时才访问飞行计划数据库。统一接口(界面)在这些情况下是一个坏主意,因为它会减慢与最常用系统的交互。然而,如果操作员与所有组成系统交互,则统一的 UI 可能是最佳方式。

3. SoS 的“开放性”。如果 SoS 是开放的,使得新系统可以在使用时被添加到其中,则统一的 UI 开发是不切实际的。不可能预见新系统的 UI 将是什么。开放性也适用于使用 SoS 的组织。如果新的组织可以参与,则他们可能根据现有的设备和他们自己的偏好用于用户交互。因此他们可能不喜欢统一的 UI。

在实践中,UI 统一的限制因素可能是 UI 开发的预算和时间。UI 开发是最昂贵的系统工程活动之一。在许多情况下,没有足够的项目预算来支持创建统一的 SoS 用户界面。

20.4.2 集成和部署

系统集成和部署通常是单独的活动。集成和测试团队将构件集成为系统,进行确认,然后发布以进行部署。构件将被管理,以便控制变更,并且集成团队可以确信系统中包含所需的版本。然而,对于 SoS,这种方法也许是不可能的。一些构件系统可能已经部署和使用,并且集成团队无法控制对这些系统的变更。

因此,对于 SoS,将集成和部署视为同一进程的一部分是有意义的。这种方法反映了将在下一节中讨论的设计指南之一,即一个不完整的系统之系统应该可用并提供有用的功能。集成过程应从已经部署的系统开始,将新系统添加到 SoS 中,以便为整个系统的功能提供一致的补充。

计划 SoS 的部署以反映这一点往往是有意义的,因此 SoS 部署发生在多个阶段。例如,图 20-8 说明了 iLearn 数字学习环境的三阶段部署过程。

1. 初始部署阶段提供身份验证、基本学习功能以及与学校管理系统的集成。

2. 部署的第 2 阶段增加了一个集成存储系统和一组更专门的工具,以支持特定主题的学习。这些工具可能包括历史档案、科学模拟系统和计算编程环境。

3. 第 3 阶段增加了用户配置的特性以及用户将新系统添加到 iLearn 环境的能力。该阶段允许为不同年龄组创建不同版本的系统,还提供了进一步的专用工具,以及要包括的标准工具的替代。

与任何大型系统工程项目一样,系统集成中最耗时和昂贵的部分是系统测试。系统之系统的测试是困难和昂贵的,有以下 3 个原因。

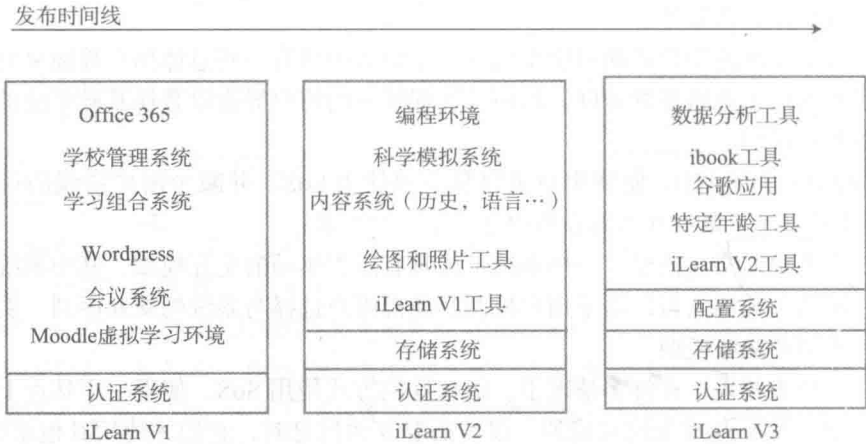


图 20-8 iLearn SoS 的发布顺序

- 1. 可能没有可用作系统测试基础的详细需求规格说明。开发 SoS 需求文档可能不具有成本效益，因为系统功能的细节由所包括的系统定义。
- 2. 组成系统可能在测试过程中发生变化，因此测试可能不可重复。
- 3. 如果发现问题，可能无法通过要求更改一个或几个组成系统来解决这些问题。相反，可能必须引入一些中间软件来解决问题。

为了帮助解决这些问题，SoS 测试应该采取下面这些由敏捷方法开发的测试技术。

1. 敏捷方法不依赖于用于系统验收测试的完整系统规格说明。相反，利益相关者密切参与测试过程，并有权决定整个系统何时可以接受。对于 SoS，如果可能的话，一些利益相关者应该参与测试过程，他们可以评论系统是否准备好部署。

2. 敏捷方法广泛使用自动化测试。这使得它更容易重新运行测试，以发现意外的系统变更是否对 SoS 整体造成了问题。

根据系统类型，你可能必须对设备安装和用户培训进行谋划，因而是部署过程的一部分。如果系统安装在新环境中，设备安装是很简单的。但是，如果要更换现有系统，假如新设备与正在使用的设备不兼容，则可能会出现安装问题。可能没有用于将新设备与工作系统并排安装的物理空间；可能没有足够的电力；可能用户没有时间参与，因为他们忙于使用当前系统。这些非技术问题可能会延迟部署过程，并使 SoS 不易被采纳和使用。

20.5 系统之系统的体系结构

系统之系统的工程过程中最关键的活动或许就是体系结构设计了。体系结构设计包含：选择要包含的子系统，评估子系统如何进行互操作，以及设计交互促进机制。体系结构设计做出了数据管理、冗余和通信方面的关键决策。本质上，由系统之系统的架构师负责实现系统概念设计中陈述的愿景。特别地，对于组织的和联邦的系统而言，这一步骤中做出的决策对系统之系统的性能、韧性和可维护性至关重要。

Maier (Maier 1998) 论述了复杂系统体系结构设计的 4 条基本原则。

- 1. 设计出的系统即使不完整也能创造价值。对于由数个其他系统组成的系统，它不应只在所有构件都正常工作的前提下才有用。应当存在一些“稳定的中间形式”，使得部分系统能够正常工作。

2. 对能控制的部分保持务实的态度。只有当个人或团体对整个系统和系统的组成部分施加控制的时候,系统之系统才能达到最佳性能。如果没有控制,想让系统之系统创造价值就相当困难。然而,试图过度控制系统之系统很可能遭到单个系统所有者的抵制,从而导致系统部署和演化方面的顺向延迟。

3. 以系统接口为重点。为构建一个成功的系统之系统,我们需要设计接口使得系统元素可以进行互操作。接口不宜具有太强的限制性,以便系统元素演化和持续发挥效用。

4. 提供协调激励机制。当系统元素被独立地拥有和管理时,系统拥有者需要激励机制来持续参与整个系统的运作。可以是财务激励(付费使用或降低经营成本)、渠道激励(如果你分享你的数据,我就会分享我的数据),或者社区激励(参与 SoS,你就能在社区得到话语权)。

Sillitto (Sillitto 2010) 完善了这些原则,并且提出了额外的重要指导原则,这些原则包含如下方面。

1. 把 SoS 设计成节点和网络体系结构。节点是一种社会技术系统,它包含了数据、软件、硬件、基础设施(技术构件)以及组织政策、人、过程和培训(社会技术的)。Web 不仅是节点间的通信设施,而且提供了每个节点上系统管理者之间正式和非正式的社交的机制。

2. 将行为指定为节点间交换的服务。面向服务的体系结构的发展提供了系统可操作性的一个标准机制。如果一个系统未提供服务接口,那么应该实现一个接口作为 SoS 开发过程的一部分。

3. 理解和管理系统缺陷。任何 SoS 都存在意外失效和不合需要的行为。尝试理解缺陷和设计一个对失效有韧性的系统是至关重要的。

从 Maier 和 Sillitto 的工作中能提取出一个关键信息: SoS 的架构师必须高瞻远瞩。他们需要将系统视为一个整体,把技术和社会技术方面都考虑进去。有时候问题最好的解决方案不是设计更多的软件,而是对管理系统运行的规则政策做出改变。

诸如 MODAF (MOD 2008) 和 TOGAF (TOGAF 是 The Open Group 2011 的注册商标) 这类的体系结构框架被认为是支持系统之系统的体系结构设计的一种方式。体系结构框架原本开发出来是为了支持企业系统体系结构的,是各个系统的组合。企业系统可以是有组织的系统之系统,也可以包含一个更简单的管理体系结构,后者使得整体管理系统组合成为可能。体系结构框架用于有组织的系统之系统的开发,在这种开发过程中整个 SoS 具有单独的治理权。

单个体系结构模型无法展示出体系结构和商业分析中所需的全部信息。因此,在框架中我们会提出数个在描述和记录企业系统的过程中应当创建和维护的体系结构视图。不同框架有太多的共同点,以致它们会倾向于反映涉及系统的语言和历史。例如,MODAF 和 DODAF 是英国国防部(MOD)和美国国防部(DOD)的类似框架。

TOGAF 框架是由 Open Group 开发的开放式标准,被用于支持企业内商业体系结构、数据体系结构、应用体系结构和技术体系结构的设计。它的核心是体系结构开发方法(Architecture Development Method, ADM),后者包含了多个阶段。这些内容如图 20-9 所示(取自 TOGAF 参考文档,Open Group 2011)。

所有的体系结构框架都包含大量的体系结构模型的开发和管理过程。图 20-8 中的每个活动都会主导系统模型的创建。然而,由于如下两个原因,这个过程可能存在问题。

1. 原始模型开发周期长,涉及项目利益相关者之间的广泛协商。这会拖慢整个系统的开发进程。

2. 维护模型的一致性非常耗时和昂贵,因为 SoS 的组织 and 内部系统经常发生变更。

体系结构框架本质上是还原论的，最大限度地忽视了技术科学和政策方面的问题。当框架意识到这些开放式的问题非常难以定义之后，会拟定出一个很多系统之系统无法达成的控制和管理标准。这个标准本质上就是一个有用的清单，提醒架构师体系结构设计过程中哪些东西是值得思考的。然而，框架采取的模式管理和还原论方法的开销限制了它们在 SoS 体系结构设计中的效用。

20.5.1 系统之系统的体系结构模式

第 6、17 和 21 章描述了不同类型的系统的体系结构模式。总的来说，体系结构模式就是不同系统公认的一种程式化的体系结构。体系结构模式在如下两个方面颇为有用：1. 促进关于系统和文献最合适的体系结构的讨论；2. 解释所采用的体系结构。本书的这一部分涵盖了软件系统之系统的多种典型模式。正如所有的体系结构模式那样，实际的系统通常基于两个或以上的模式。

系统之系统的体系结构模式的概念还在起步阶段。Kawalsky (Kawalsky 等, 2013) 探讨了体系结构模式在理解和支持 SoS 设计中的价值，尤其是指挥控制系统。模式在描述 SoS 组织的时候非常有效，无须详细的领域知识。

作为数据提要的系统

在此类体系结构模式中 (见图 20-10)，存在一个要求不同数据类型的主系统。数据可从其他系统中获取，主系统查询其他系统以获得所需的数据。总体而言，提供数据的系统之间不存在交互。这种模式常见于组织型系统或联邦型系统，因为这些系统中存在着一些治理机制。

例如，在英国，要想给车上牌照，必须首先提供有效保险和年检证书。当你和车辆管理系统打交道的时候，系统其实已经和两个其他系统进行了交互，以确保提供的证书文件是有效的。两个其他系统包括：

1. 投保车辆系统。一个由车辆保险公司运行的联邦型系统，维护所有最新的车辆保险政策的信息。
2. MOT 证书系统。记录所有国家权威检测机构颁发的年检证书。

“作为数据提要的系统”这一体系结构，在可唯一验证实体并针对这些实体创建相对简单的查询操作的情况下，是适合使用的。在牌照系统中，车辆由车牌注册号唯一标识。在其他系统中，可以通过 GPS 坐标定位如污染监控仪之类的实体。

“作为数据提要的系统”的变体产生的原因是，一定数量的系统提供相似的数据，而非完全一致的数据。因此，系统不得不引入中间层，如图 20-11 所示。中间层的作用是把来自自主系统的查询语言翻译成各个子信息系统所需要的特定查询语言。

例如，iLearn 系统和 3 个不同的供应商提供的学校管理系统交互。所有的学校管理系统

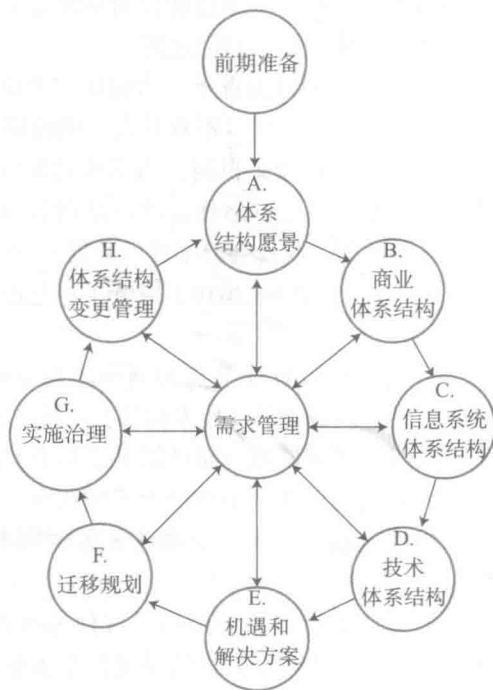


图 20-9 TOGAF 体系结构开发方法 (TOGAF® Version 9.1, ©1999-2011.The Open Group)

提供学生的相同信息（姓名、个人信息等），但是拥有不同的接口。数据库的组织结构各异，返回的数据类型也各不相同。统一接口使用区域性信息检测系统用户基于哪个区域，从而得知应当访问哪个管理系统，然后将标准查询转化为该系统的特定查询。

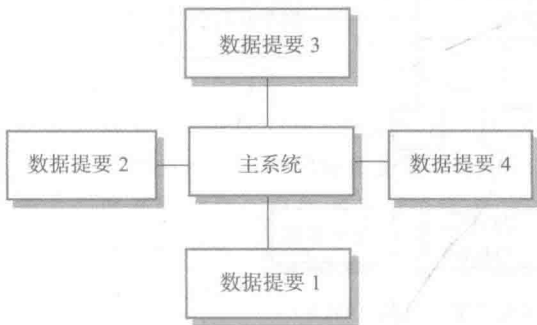


图 20-10 作为数据提要的系统

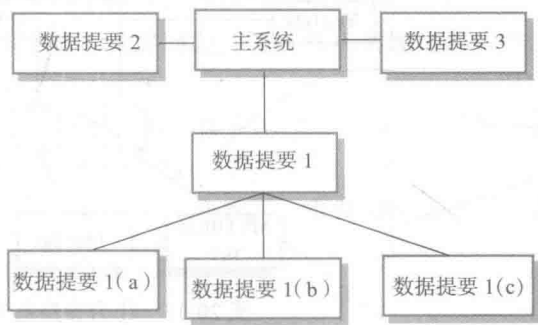


图 20-11 具有统一接口的作为数据提要的系统

此模式的系统中出现的问题本质上是数据提要无法获取或响应慢导致的接口问题。系统中应当有超时设定，以避免数据提要失效影响整个系统的响应时间。同时还应有治理机制，以防止在未取得所有系统所有者的允许的情况下修改提供数据的格式。

容器中的系统

容器中的系统是系统之系统的一种，其中一个系统承担虚拟容器的作用，提供诸如身份验证和存储的公共服务。概念上而言，其他系统被放入这个容器，以确保用户能使用它们的功能。图 20-12 描述了一个拥有 3 个公共服务和 6 个内置系统的容器系统。包含的内置系统是从核准系统列表中选取的，并且无须知道它们被包含在容器内。这种模式常见于联邦型系统或系统联合中。

iLearn 系统就是一种容器中的系统，拥有支持身份验证、用户数据存储和系统设置等功能的公共服务。其他功能通过选取现存的系统（包括报纸存档、学习环境等）并集成到容器内实现。

当然，你不必把这些系统置入一个真实的容器来实现系统之系统。对于每一个核准的系统，都有一个特定的接口来实现其与公共服务的集成。接口管理容器提供的公共服务的转化和已集成系统的需求。集成未经核准的系统也是可能的，只是这些系统无法访问容器提供的公共服务。



图 20-12 容器中的系统

图 20-13 描述了这种集成的过程。这幅图是提供了如下三种公共服务的 iLearn 系统的简化版本。

1. 一个提供了所有核准系统上的登录功能的验证服务。用户不必为不同的系统维护不同的证书。
2. 用户数据存储服务。这项服务可以无缝地转移到核准系统，或从核准系统转出。
3. 配置服务。用以在容器内增加或删除系统。

这个案例描述了针对物理学科的 iLearn 系统的一个版本。除了办公效率系统（如 Office 365）和虚拟学习环境（如 Moodle），还包含了模拟器和数据分析系统。其他系统（如 YouTube 和科学百科全书）也是这个系统的一部分。然而，这些是未经核准的，因此也不具备容器接口。用户必须亲自登入这些系统来组织数据的传输。

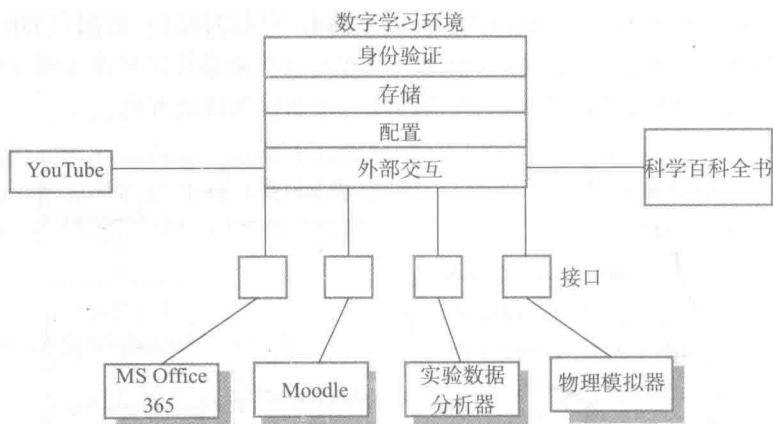


图 20-13 作为容器系统的数字学习环境 (DLE)

此类 SoS 体系结构存在两个问题。

1. 必须为每个核准系统开发一个单独的接口，以保证公共服务能正常使用。这意味着只能支持相对少数的核准系统。

2. 容器系统的拥有者无法对内置系统的功能和行为施加影响。内置系统随时可能发生故障或者被撤出容器。

然而，这个体系结构的主要好处在于它允许增量式开发。容器系统的早期版本可以基于未经核准的系统。接口可以在后面的版本开发，以便其与容器服务更紧密地集成。

交易系统

交易系统是系统之系统的一种，它不包含主系统，任何成分系统之间都可能发生信息的交易，可能是一对一，也可能是一对多。每个系统发布自己的接口，但不存在所有系统都遵循的接口标准。这样的系统如图 20-14 所示。交易系统可能是联邦型系统或系统联合。

交易系统的一个例子就是一个用于股票证券的算法交易的系统之系统。经纪人拥有自己的系统，可以自动地从其他系统中买卖股票。这些系统可以定价，也可以和其他系统单独协商。交易系统的另外一个例子就是旅行聚合器，可以显示价格进行比较，并且允许用户直接预订旅行。

交易系统通常为某种类型的集市而开发，其中的信息交换大多是关于交易的商品和商品价格的。尽管交易系统有自己的权限，在个人交易中很有说服力，但是主要还是用于自动交易场景，在这种场景下，系统之间直接进行协商。

这种模式的主要问题是缺乏治理机制，内置系统可以随时变更。当这种变更和其他系统的预设矛盾的时候，交易就无法继续。有时，联盟中的系统的拥有者希望能与其他系统继续交易，因此他们可能做出一些非正式的安排来确保其中一个系统的变更不会阻止交易的继续进行。在其他情况下，如旅行聚合器中，航空公司可能故意改变系统来强制用户直接和他们进行预订。

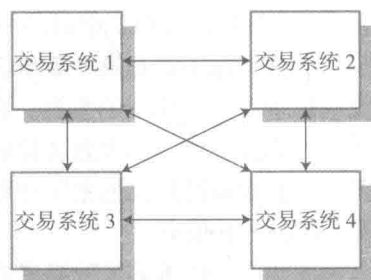


图 20-14 一个交易系统之系统

要点

- 系统之系统是两个或以上独立管理和治理的系统的集合。

- 系统之系统有3种重要的复杂性类型——技术复杂性、管理复杂性和治理复杂性。
- 系统治理可以作为 SoS 的分类方法的基础。这引出了 SoS 的3个分类——组织型系统、联邦型系统和系统联合。
- 由于系统之系统内部的复杂性，还原论这一工程方法在这里并不适用。还原论假定有清晰的系统边界、理性的决策制定和明确定义的问题。这3条在系统之系统中都不成立。
- SoS 开发过程中的关键步骤包括概念设计、系统选择、体系结构设计、接口开发、集成与部署。治理和管理政策必须与这些活动同步设计。
- 体系结构模式是描述和讨论 SoS 的典型体系结构的一种方式。重要的模式包括：作为数据提要的系统、容器中的系统和交易系统。

阅读推荐

《Architecting Principles for Systems of Systems》是现今关于系统之系统的一篇经典论文，介绍了 SoS 的分类方法，探讨了它的价值，并提出了系统之系统设计的一些体系结构原则。(M. Maier, Systems Engineering, 1(4), 1998)

《Ultra-large Scale Systems: The Software Challenge of the Future》这本书于2006年为美国国防部创作，介绍了拥有数百个节点的超大规模系统的概念，探讨了开发此类系统过程中的问题和挑战。(L. Northrop et al., Software Engineering Institute, 2006) <http://www.sei.cmu.edu/library/>

《Large-scale Complex IT Systems》这篇论文讨论了作为系统之系统的大规模复杂IT系统的一些问题，拓展了关于还原论失败的观点，提出了 SoS 研究领域的一些挑战。(I. Sommerville et al., Communications of the ACM, 55(7), July 2012) <http://dx.doi.org/10.1145/2209249.2209268>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap20/>

支持视频的链接: <http://software-engineering-book.com/videos/systems-engineering/>

练习

- 20.1 为什么当比较系统之系统和其他大型复杂系统的时候管理和运营的独立性是关键的区别因素？
- 20.2 治理复杂性和管理复杂性之间的区别是什么？
- 20.3 20.2 节讲述的 SoS 的分类中提出了一个基于治理的分类方法。指出下列系统之系统的分类，并给出你的理由：
 - (a) 一个给医院、诊所和基层医疗的患者健康记录提供统一访问权限的医疗保健系统；
 - (b) 万维网；
 - (c) 一个向一系列福利服务（如养老金、伤残抚恤金和失业津贴）提供访问权限的政府系统。
- 20.4 什么是还原论？为什么它能作为多种工程的基础？
- 20.5 使还原论不那么有效的复杂软件系统有什么特征？

- 20.6 为什么在 SoS 开发过程中系统选择和体系结构设计应并发执行？为什么两者之间应该有紧密的联系？
- 20.7 Sillitto 认为 SoS 中的节点之间的通信不应该只是技术方面的，还应该包含系统涉及的人之间的非正式社会技术交流。使用 iLearn 系统之系统作为例子，指出在哪些方面非正式交流能提高系统有效性。
- 20.8 指出练习 20.3 中的系统之系统最匹配的体系结构模式。
- 20.9 交易系统模式假定没有中央集权的情形存在。然而在诸如资本交易等领域，交易系统必须遵循规范性规则。指出为允许监管者检查是否遵循规则，这个模式应做出哪些调整。不包含所有经过中心节点的交易。
- 20.10 你就职的软件公司开发过一个提供购买者信息的系统，并且已经被一部分零售企业在 SoS 内使用。他们为使用的服务付费。讨论在不事先通知的情况下改变系统接口以强迫用户支付更高的费用有何种道德衡量。分别从企业员工、顾客和股东的角度考虑这个问题。

参考文献

- Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.
- Hitchins, D. 2009. "System of Systems—The Ultimate Tautology." <http://www.hitchins.net/profs-stuff/profs-blog/system-of-systems---the.html>
- Kawalsky, R., D. Joannou, Y. Tian, and A. Fayoumi. 2013. "Using Architecture Patterns to Architect and Analyze Systems of Systems." In *Conference on Systems Engineering Research (CSER 13)*, 283–292. doi:10.1016/j.procs.2013.01.030.
- Maier, M. W. 1998. "Architecting Principles for Systems-of-Systems." *Systems Engineering* 1 (4): 267–284. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.
- MOD, UK. 2008. "MOD Architecture Framework." <https://www.gov.uk/mod-architecture-framework>
- Northrop, Linda, R. P. Gabriel, M. Klein, and D. Schmidt. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh: Software Engineering Institute. http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf
- Open Group. 2011. "Open Group Standard TOGAF Version 9.1." <http://pubs.opengroup.org/architecture/togaf91-doc/arch/>
- Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.
- Royal Academy of Engineering. 2004. "Challenges of Complex IT Projects." London. <http://www.bcs.org/upload/pdf/complexity.pdf>
- Sillitto, H. 2010. "Design Principles for Ultra-Large-Scale Systems." In *Proceedings of the 20th International Council for Systems Engineering International Symposium*. Chicago.
- Sommerville, I., D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. McDermid, and R. Paige. 2012. "Large-Scale Complex IT Systems." *Comm. ACM* 55 (7): 71–77. doi:10.1145/2209249.2209268.
- Stevens, R. 2010. *Engineering Mega-Systems: The Challenge of Systems Engineering in the Information Age*. Boca Raton, FL: CRC Press.

实时软件工程

目标

本章的目标是介绍嵌入式实时系统的重要特征和实时软件工程。阅读完本章后，你将：

- 理解嵌入式软件的概念，嵌入式软件用于必须对那些来自其环境的外部事件做出响应的控制系统；
- 了解实时系统的设计过程，即软件系统组织成一组协作的过程；
- 理解常用于嵌入式实时系统设计的三大体系结构模式；
- 理解实时操作系统的组织结构以及它们在嵌入式实时系统中的角色。

从简单的家用电器到游戏控制器再到整个制造工厂，采用计算机控制的系统非常广泛。这些计算机直接与硬件装置交互。它们的软件需要对硬件产生的事件做出响应并发出相应的控制信号。这些信号引发动作，例如，开始一个电话呼叫、在屏幕上角色的移动、阀门的打开、系统状况的显示。这些系统中的软件嵌入硬件中，通常是只读存储器里，并且总是能实时地响应系统环境中的事件。所谓实时，是指软件系统响应外部事件有一个时限。如果超过了这个时限，整个硬件软件系统将不能正确地运行。

现在由于几乎每种电器设备都包括软件，嵌入式软件变得非常重要。与其他类型的软件系统相比，嵌入式软件系统更多。根据 Ebert 和 Jones (Ebert and Jones 2009) 的估算，发达国家人均拥有约 30 个微处理器系统，这一数字还在以每年 10% ~ 20% 的速率增长。这预示着到 2020 年，人均嵌入式系统拥有量将超过 100 个。

实时响应是嵌入式系统和其他软件系统之间的一个最为重要的不同点。这里所说的其他系统，如信息系统、基于 Web 的系统或者个人软件系统，它们的主要目的是数据处理。对于非实时系统，它们的正确性可以通过指定系统输入如何映射到由系统产生的对应输出来定义。对一个输入的响应，系统应该产生对应的一个输出，通常应该存储一些数据。例如，在病人信息系统中，如果你选择一条创建命令，那么正确的系统响应是在数据库中创建一条新的病人记录，并证实这已经实现。在合理的限度内，它的执行与时间无关。

然而，在实时系统中，准确性既依赖于对输入的响应，又依赖于产生该响应的的时间。如果系统要花费太长时间去响应的话，那么所要的响应可能是无效的。例如，如果嵌入式软件控制的汽车刹车系统执行太慢，那么由于车不能及时停下来，就会发生事故。

因此，在实时软件系统的定义中，时间是固有要素：

实时系统是一个软件系统，系统的功能正确与否取决于系统产生的结果以及产生这个结果的时间。“软”实时系统是这样一个系统，如果结果没有在规定的时间内产生，则它的操作被视为退化的操作。“硬”实时系统是这样一个系统，如果结果没有在规定的时间内产生，则它的操作被视为系统失效。

对于所有嵌入式系统来说，及时响应是一个重要的因素，但是，在某些情况下，却不需要非常快速的响应。例如，在前文中使用过的胰岛素泵系统就是一个嵌入式系统。然而，在

它需要对血糖值做检测的周期性间隔当中，不需要对外部事件进行十分迅速的响应。野外气象站软件也是一个嵌入式系统，但是它也不要求对外部事件进行迅速的响应。

除了包含实时响应的需求以外，嵌入式系统和其他类型的软件系统之间还有其他的重要不同点。

1. 嵌入式系统通常持续运行而不终止。当硬件开启时，系统开始执行，且必须执行直到硬件被关闭。这就意味着可靠的软件工程技术（第 11 章中介绍的）必须连续运行。实时系统可能包括支持动态重配置的更新机制，这样系统处于服务期间也能得到更新。

2. 与系统环境的交互是不可控制和不可预测的。在交互系统中，交互的节奏是受系统控制的。通过限制用户选择，待处理的事件是可以提前了解的。相反，实时嵌入系统必须能对任何时候的意外事件做出响应。这导致了对实时系统的设计是基于并发性的，多个进程并行执行。

3. 存在影响系统设计的物理限制。这样的例子包括系统可利用的电源限制和硬件占据的物理空间的限制。这些限制可能会产生对嵌入式软件的需求，例如，节约能量以延长电池供电时间的需求。尺寸和重量的限制意味着软件要承担一些硬件功能，因为系统需要限制使用的芯片数量。

4. 直接硬件交互是必要的。在交互式系统和信息系统中，有一个软件层（设备驱动）对操作系统隐藏了硬件。这是可能的，因为你只能将几种类型的设备连接到系统，例如键盘、鼠标、显示器等。相反，嵌入式系统要和各种各样的硬件设备交互，而这些设备没有单独的设备驱动。

5. 安全性和可靠性在系统设计中占据主要的地位。许多嵌入式系统控制的设备如果失败了会带来高昂的人力或经济成本。因此，可依赖性是非常重要的，并且系统设计必须始终保证安全临界行为。通常采用保守的方法设计，即使用已调试或已测试的成熟技术，而不是使用可能会导致新失效模式的新技术。

嵌入式系统可以看作是反应式系统，即它们必须在其环境中对事件做出反应（Berry, 1989；Lee, 2002）。响应时间通常是由物理学定律所控制的，而非为人的便捷所选择的。这一点与其他软件系统很不一样，其他类型软件系统控制着交互速度。例如，用于写书的字处理器能够检查拼写和语法，对处理所用的时间没有限制。

21.1 嵌入式系统设计

嵌入式系统的设计过程是一个系统工程的过程，在该过程中软件设计者要深入系统能力考虑系统硬件的设计和性能。部分系统设计过程是要决定，哪些系统能力要用软件实现，哪些系统能力要用硬件实现。对于很多嵌入在消费类产品（如手机）中的实时系统来说，硬件的成本和电能消耗是一个非常重要的问题。需要使用专用的处理器来支持嵌入式系统，而对于某些系统，可能还需要设计和建造专用的硬件。

这意味着自上而下的软件设计过程，即设计开始于通过一系列阶段对抽象模型进行分解和开发的设计过程，对于绝大多数实时系统来说是不现实的。对于硬件、支持软件以及系统时序等方面的一些下层决策，需要在过程的早期考虑。这些限制了系统设计者的灵活性，可能意味着一些额外的软件功能，比如电池和电源管理必须包括在系统内。

考虑到嵌入式系统是对它们的环境中的事件做出反应的反应式系统，对嵌入式实时软件设计的一般方法是基于激励-响应模型。激励是发生在软件系统环境中引起系统以某种方式

响应的事件；响应是由软件发送到它的环境的一种信号或消息。

我们可以通过列出由系统接收的激励和相关联的响应，以及响应必须产生的时间，来定义一个实时系统的行为。例如，图 21-1 给出了防盗报警系统的可能的激励和系统响应。21.2.1 节给出了更多关于该系统的信息。

激 励	响 应
清除警报	关掉所有开启的警报；关掉所有打开的照明灯光
控制台紧急按钮被按下	启动警报；打开控制台附近照明灯光；呼叫警察
电源供电失效	报告服务技术人员
传感器失效	报告服务技术人员
单个传感器结果为正	启动警报；打开此传感器附近的照明灯光
两个或更多的传感器结果为正	启动警报；打开此传感器附近的照明灯光。报告警察有可疑闯入者的位置
电压下降 10% ~ 20%	切换到备份电池供电；运行电源供电测试
电压降幅超过 20%	切换到备份电池供电；启动警报；报告警察；运行电源供电测试

图 21-1 防盗报警系统的激励和响应

激励有两类：

- 1. 周期性的激励。在每个可预测的时间间隔内发生。举例来说，系统可能每隔 50ms 检查一次传感器，根据传感器的值（激励）采取响应行动。
- 2. 非周期性的激励。这种激励是不规则发生的，也是不可预测的。通常使用计算机的中断机制发送信号。这种激励的一个例子是中断，它指示输入 / 输出传输完成且数据已经在缓冲器中。

如图 21-2 所示，激励来自于系统环境中的传感器，响应被发送到执行器（actuator）。这些执行器控制设备，例如泵，由此对系统环境做出改变。执行器自身也可以产生激励。来自于执行器的激励通常意味着执行器出现了一些问题，必须由系统处理。

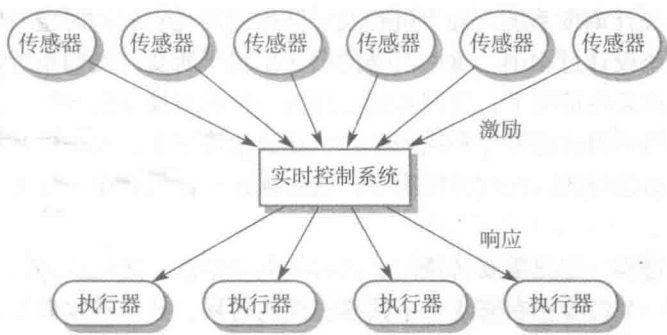


图 21-2 嵌入式实时系统的一般模型

实时系统的一般设计准则是，对于每一类传感器和执行器有不同的进程（见图 21-3）。对每一类传感器，有一个传感器管理进程处理由传感器采集的数据。数据处理进程计算系统接收到激励所需要的响应。执行器控制进程与每个执行器相关联，并且管理执行器的操作。传感器和执行器进程这个模型能从传感器迅速采集数据（在被下个输入覆盖之前），并允许之后对数据的加工以及相关执行机构的响应执行。

实时系统需要响应不同时间发生的激励，因此，它的体系结构的组织需要保证只要接收到激励，立即将相应的控制传到适当的处理单元中去。这在顺序程序中是不现实的，因此，正常情况下，实时软件系统都被设计成一组并发协作的进程。为了支持对这些进程的管理，实时系统所处的执行平台可能包括实时操作系统（将在 21.4 节讨论）。实时操作系统所提供的功能可通过实时支持系统进行访问。



图 21-3 传感器和执行器进程

并没有标准的嵌入式系统设计过程。使用何种过程取决于系统的类型、可利用的硬件以及开发系统的机构。实时软件设计过程可能包含以下几步。

1. 平台选择。此活动中，为系统选择一个执行平台（即要使用的硬件和实时操作系统）。影响这些选择的因素包括系统的时序限制、可用电源的限制、开发团队的经验、交付系统的价格目标等。

2. 激励 / 响应识别。包括识别系统必须处理的激励和针对每种激励相应的一个或多个响应。

3. 时序分析。对于每种激励和相应的响应，找到应用于激励和响应处理的时序约束。依据这些约束在系统中为进程建立时限。

4. 进程设计。在这个阶段，将激励和响应处理聚集到几个并发的进程中。设计进程体系结构的好的起点是如 20.2 节所描述的体系结构模式。然后，是对进程体系结构的优化以反映出你需要实现的特殊需求。

5. 算法设计。对于每个激励和响应，设计算法以执行所需要的运算。算法设计需要在设计过程中相对早些进行，给出要求处理的数量的提示和完成处理所需要的时间的提示。这对于计算密集型任务（如信号处理）是相当重要的。

6. 数据设计。定义进程交换的信息以及协调信息交换的事件，并且设计相应的数据结构管理信息交换。几个并发进程可以共享这些数据结构。

7. 进程调度。设计调度系统，确保进程及时开始以满足它们的时限要求。

在实时软件系统设计过程中，这些活动的顺序依赖于所要开发的系统的类型、它的过程和它的平台需求。在某些情况下，我们可能会遵循一种相对抽象的方法，即从激励和相关联的处理开始，并在稍后再决定硬件和执行平台。在其他情形下，对硬件和操作系统的选择是在软件设计开始之前进行的，在这种情况下，就必须在软件设计中充分考虑硬件能力所带来的约束。

实时系统中的进程一定是需要协调的，也需要共享信息。进程协调机制要确保在共享资源时相互排斥。当一个进程正在修改某个共享资源的时候，其他进程就不能改变该资源。确保相互排斥的机制包括：信号灯机制，监视器机制，关键区域机制。在操作系统的教材中有关于这些过程的同步机制的详细描述（Silberschatz, Galvin, and Gagne 2013；Stellings 2014）。

当设计进程间信息交换时，要考虑这些进程以不同的速度在运行这一事实。一个进程产生信息，另一个进程消费那个信息。如果生产者比消费者运行得快，在消费者进程读原来的信息之前，新的信息可能重写先前读入的信息项。如果消费者进程比生产者进程运行快，同样的信息可能被读两遍。

为了解决这个问题，应该使用共享缓冲区并且利用互斥机制控制对缓冲区访问，以便实现信息交换。这就意味着在信息被读之前不可能被重写，并且信息不能被读两次。图 21-4 展示了共享缓冲区的概念。这通常作为循环队列实现，这样生产者和消费者进程之间的速度不匹配就会得到调节，不至于耽误进程执行。

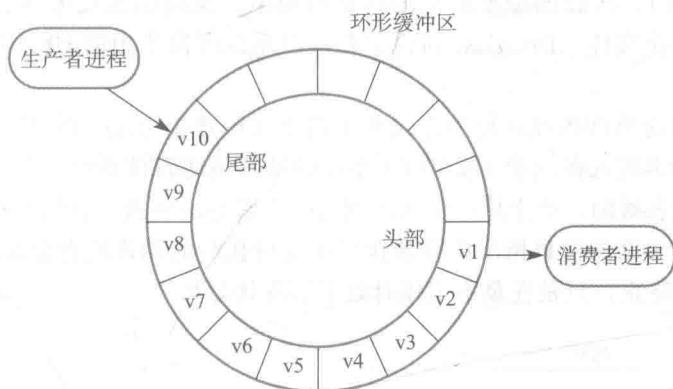


图 21-4 生产者/消费者进程共享循环缓冲区

生产者进程一直在缓冲区队列尾部位置存入数据（在图 21-4 中以 v10 表示）。消费者进程一直从队列头部检索信息（在图 21-4 中以 v1 表示）。消费者进程检索信息之后，列表的尾部被调整到下一项（v2）处。当生产者进程增加信息后列表尾部调整到列表的下一空位处。

很明显，确保生产者和消费者进程不同时访问同一项（即 Head=Tail 时）是非常重要的。如果二者同时访问同一项，该对象的值可能难以预料。必须确保生产者进程不能向满缓冲区添加项，消费者进程不能从空缓冲区取出项。

为了实现这一点，将循环缓冲区实现为一个进程，采用 Get 和 Put 操作访问缓冲区。生产者进程调用 Put 操作，消费者进程调用 Get 进程。同步原语，如信号量或者关键区域，被用于确保 Get 和 Put 操作是同步的，这样它们不会在同一时间访问同一位置。如果缓冲区满了，Put 进程就等待一直等到一个槽变空；如果缓冲区空，Get 进程一直等待直到产生了一个条目。

一旦为系统选择了执行平台，设计了一个进程体系结构并确定了调度策略，接下来就要检查系统是否能满足它们的时序需求。我们可以根据构件的时序行为知识通过静态分析来做此项工作，也可以通过仿真来完成。这样的分析能暴露出系统不能很好地运行的情形。进程的体系结构、调度策略、执行平台或者所有这些都必须重新设计，以便提高系统的性能。

时序约束或者是其他需求有时意味着我们最好是用硬件而不是软件来实现某些系统功能，比如信号处理。最新的硬件构件，例如 FPGA（Field-Programmable Gate Array），它是灵活的，可以适应不同的功能。硬件构件较之软件构件来说能带来更好的性能。可以找到系统处理瓶颈并采用硬件来更换，这样能避免昂贵的软件优化。

21.1.1 实时系统建模

实时系统需要响应的事件通常引起系统从一个状态转换到另一个状态。因为这个原因，如第 5 章描述的，状态模型通常用来描述实时系统。系统的状态模型假设为，系统在任一时

刻一定处于多个可能状态的某个状态中。当接收到一个激励时，系统可能就会产生一次状态转换。举例来说，当操作人员给出命令（激励）时，控制阀门的系统可能从“阀门开启”状态转换到“阀门关闭”状态。

状态模型是实时系统设计方法的一个完整部分。UML 使用状态图支持状态模型的开发（Harel 1987, 1988）。状态图是形式化的状态机模型，支持层次化的状态，这样多个状态组可以当成一个单个实体。Douglass 讨论了在实时系统开发当中的 UML 的使用（Douglass 1999）。

第 5 章已经用简单的微波炉模型来说明了这个系统建模方法。图 21-5 是另一个状态机模型的例子，它给出嵌入在汽油（或燃气）泵中的油料输送软件系统的操作。圆角矩形代表系统状态，箭头代表激励，产生从一个状态到另一个状态的转换。在状态机图中所选择的名称是描述性的，相关联的信息指出由系统执行单元所执行的动作或者是显示的信息。注意，这个系统从来不会终止，只是在泵不工作时处于等待状态。

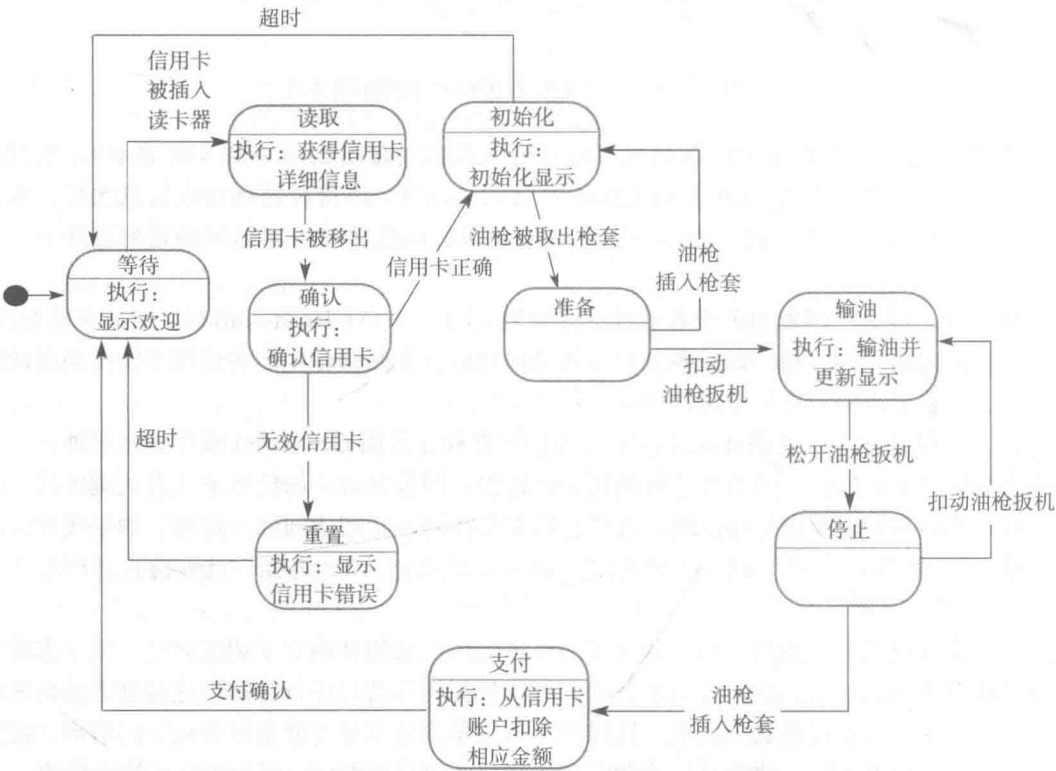


图 21-5 汽油（燃气）泵的状态机模型

- 油料传递系统是为无看守运行而设计的，具有以下动作顺序。
1. 买家将信用卡插入油泵内嵌的读卡器中，这一操作使系统转入读取状态，在该状态下，系统读取信用卡详细信息，随后系统会要求买家取回信用卡。
 2. 取出信用卡会触发系统转入验证状态，在该状态下，系统对信用卡进行验证。
 3. 如果信用卡是有效的，系统会初始化油泵，当油枪从枪套中取出时，系统转入输油状态，准备输油。扣动油枪上的扳机会开始输油，直到松开扳机时停止（为了简化描述，我忽略了防止油料溢出的压力开关）。



实时 Java

Java 编程语言已被修改，以使其适合于实时系统开发。这些修改包括异步通信、添加时间（包括绝对和相对时间）、线程不能被垃圾回收程序中断的新线程模型，以及一种新的内存管理模型，其可以避免由垃圾回收引起的不可预测的延迟。

<http://software-engineering-book.com/web/real-time-java/>

4. 在输油完成并且买家将油枪放回枪套后，系统转入支付状态。在该状态下，用户账户被扣除相应金额。

5. 支付完成后，油泵软件返回等待状态。

状态模型在模型驱动的工程中被用于定义系统操作，这一部分我在本书第五章论述过。它们可以被自动或半自动的转化成一个可执行程序。

21.1.2 实时编程

用于实时系统开发的程序语言必须包括能访问系统硬件的能力，而且它要能够预测语言中的特别操作的时序。运行在受限硬件上的硬实时系统现在有时还仍然使用汇编语言来编程，这是因为，只有使用汇编语言才能满足苛刻的时限要求。系统编程语言，比如 C 语言，能生成高效的代码，所以也被广泛使用。

使用像 C 语言这样的系统编程语言的优势是，它能够开发出高效的程序。然而，这些语言不包含支持并发或资源共享管理的结构。并行性和资源管理是通过调用实时操作系统提供的原语实现的，例如互斥信号量。而这种对实时操作系统的调用是不能被编译器检查的，所以，很容易产生编程错误。而且，这种程序通常也是很难理解的，因为语言中并不包含实时特征。在理解程序的同时，程序阅读者还要知道如何使用系统调用提供对实时性的支持。

因为实时系统必须满足它们的时序约束，对于硬实时系统就不能使用面向对象的开发。面向对象开发会通过对象中所定义的操作隐藏数据表示和访问的属性值。这意味着在面向对象系统中有非常大的开销，因为需要额外的代码协调属性的访问和处理对操作的调用。由此在性能上的损失使之不可能满足实时的时限要求。

已经有了为嵌入式系统开发所设计的 Java 版本（Burns and Wellings 2009；Bruno and Bollella 2009）。该语言包含了已修改的线程机制，允许所指定的线程不被语言垃圾收集机制所中断。也包括异步事件处理机制和时序规格说明。然而，在写作本书之际，该规格说明大多用于有强大处理器和存储能力的平台（例如手机），而不是仅有有限资源的简单嵌入式系统。后者通常仍是用 C 语言实现的。

21.2 实时软件体系结构模式

体系结构模式是对好的设计实践的抽象的类型化的描述。它们封装的知识包括：系统体系结构的组成，什么时候应该使用这些体系结构，以及它们的优势与劣势。然而，我们不应该把体系结构模式作为一种通用的设计来实现；而是应该通过模式来理解体系结构，并作为创建自己的专门的体系结构设计的起点。

正如所期待的那样，实时软件和交互式软件的不同意味着，对于嵌入式实时系统我们需要使用不同的体系结构模式。实时系统模式是面向过程的，而不是面向对象或者面向构件的。在这一节将讨论 3 种经常使用的实时体系结构模式。

1. 观察和反应模式。该模式用于当一组传感器周期性地监控和显示的时候。当传感器显示已经发生了某个事件时（如电话来电），作为反应系统会启动一个进程来处理此事件。

2. 环境控制模式。该模式用于某些系统，这些系统包括能提供环境信息的传感器和改变环境的执行器。根据传感器检测到的环境改变，控制信号被发送到系统的执行器。

3. 处理管道模式。该模式用于在数据被处理之前需要从一种表示变换到另一种表示的时候。变换实现为一系列可并行执行的处理步骤。由于单独的核或处理器可以执行每个变换，因此该模式允许非常快速的数据处理。

当然这些模式可以结合使用，并且通常可以看到，在一个系统中存在多种模式。例如，当使用环境控制模式时，非常普遍的是对于待监控的执行器使用观察反应模式。在执行器失效的事件中，系统可以通过显示一条警告信息、关闭执行器和切换到备份系统等对此事件做出反应。

此处讨论的模式是描述嵌入式系统的总体结构的体系结构模式。Douglass（Douglass 2002）描述了用于帮助编程者做出更详细设计决策的更低层的实时系统设计模式。这些模式是关于执行控制、通信、资源分配、安全性和可靠性等方面的。

这些体系结构模式是嵌入式系统设计的起点；然而它们并不是设计模板。如果当成模板使用，就会设计出低效的过程体系结构。因此，需要优化过程体系结构，从而确保系统没有太多的过程。我们也应该确保系统中的过程、传感器和执行器之间有一个清晰的联系。

21.2.1 观察和反应模式

监控系统是嵌入式实时系统的一个重要类别。监控系统通过一组传感器来检查它的环境，并且通常以某种方式显示环境的状态。这可以是在内置的屏幕上显示，或者是在特定仪器显示器上显示或者是远程显示。如果系统检测到异常事件或异常传感器状态，监控系统就会采取某些行动。通常会产生一个警报提醒操作员注意所发生的事件。有时系统可以启动一些其他的预防动作，例如关闭系统以阻止其受到损害。

观察和反应模式（见图 21-6 和图 21-7）是常用于监控系统的模式。观测传感器的值，根据这些值，系统以某种方式做出反应。监控系统由多个观察和反应模式的实例组成，系统中的每种传感器都有一个模式的实例。依据系统的需求，我们可以通过合并进程来优化设计（例如，可以用单个显示进程显示来自所有不同类型的传感器的信息）。

名称	观察和反应
描述	对一组相同类型的传感器的输入值进行收集和分析。这些值以某种方式显示出来。如果传感器的值表示有某个异常状况发生，那么就会启动一个行动来提醒操作员注意，在某些情况下，还会采取某种行动来响应这个异常值
激励	连接到系统的传感器的值
响应	到显示器的输出，报警器触发，到反应系统的信号
进程	观察者 (Observer)，分析 (Analysis)，显示 (Display)，警报 (Alarm)，反应器 (Reactor)
使用场合	监控系统、警报系统

图 21-6 观察和反应模式

这里给出一个使用该模式的例子，考虑将安装在大厦中的防盗报警系统的设计。

准备实现一个软件系统，该软件系统是安装在商务大厦中的防盗报警系统的一部分。它使用多种不同类型的传感器。包括：每个房间中的运动传感器，检查走廊门开启的门传感器，安装在一楼的能检测窗户被破坏的窗传感器。

当传感器检测到入侵者存在时，系统自动呼叫当地警察局，使用声音合成器报告警报的位置。系统打开活动传感器周围的房间内的灯，并发出声音警报。传感器系统使用主电源供电但也配备一个备用电池。使用一个单独的电力电路监视器监视电路的电压，从而发现掉电现象。在电压降低到某个水平时触发报警系统。

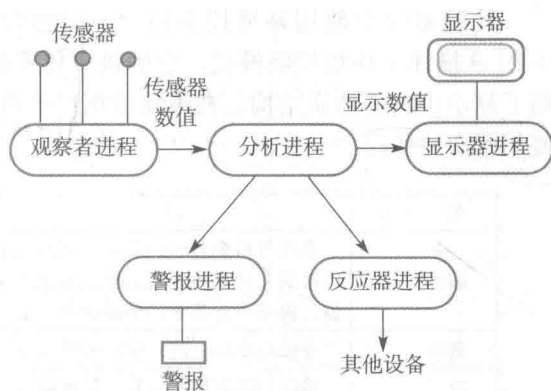


图 21-7 观察和反应的进程结构

图 21-8 显示了报警系统的进程体系结构。图中，箭头表示从一个进程发送到另一个进程的信号。这个系统是一个没有严格时序要求的“软”实时系统。传感器不需要检测高速事件，只需要检测是否有人出现，因此每秒只需要轮询它们 2 ~ 3 次。21.3 节将介绍该系统的时序需求。

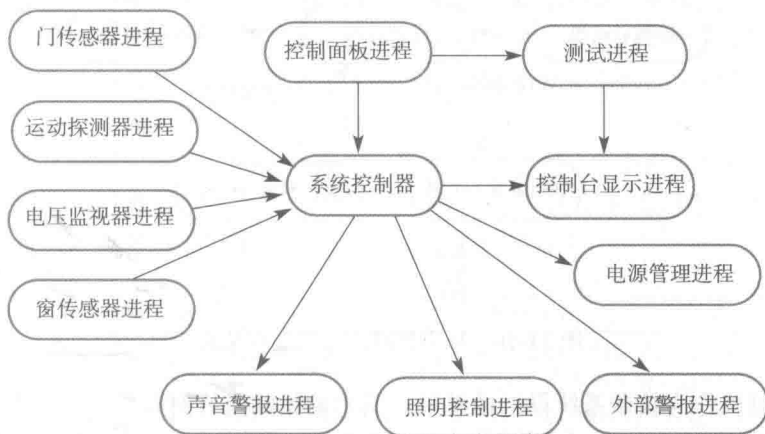


图 21-8 防盗报警系统的进程结构

在图 21-1 中已经介绍了报警系统的激励和响应。这些响应是系统设计的起点。在设计中使用了观察和反应模式。存在与每种传感器相关联的观察者进程和与每种反应类型相关联的反应者进程。还有一个单个分析进程检查来自所有传感器的数据。模式中的所有显示进程被合并为一个显示进程。

21.2.2 环境控制模式

实时嵌入式软件最为广泛的使用可能是在控制系统中。在这些系统中，软件基于来自设备所处的环境的激励来控制设备的运行。例如，汽车防滑刹车系统监控汽车的轮子和刹车系

统（系统的环境），观察给予制动压力时轮子打滑的信号。如果出现这种情况，系统调整制动压力，阻止车轮抱死，从而降低侧滑的可能性。

控制系统会使用环境控制模式，即包含传感器进程和执行器进程的通用控制模式。图 21-9 描述了环境控制模式，它的进程体系结构如图 21-10 所示。该模式的一个变种是省略了显示进程的体系结构。在不要求用户干预或控制频率很高以至于显示无意义的情况下，使用此变种模式。

名 称	环境控制
描述	系统分析来自一组采集系统环境数据的传感器的信息。进一步的信息也可能是来自连接到系统的执行器的状态信息。基于传感器和执行器的数据，传送控制信号到执行器，进而引起系统环境的改变。关于传感器的值和执行器的状态的信息会显示出来
激励	连接到系统上的传感器的值以及系统执行器的状态
响应	给执行器的控制信号，显示信息
进程	监视器 (Monitor)，控制 (Control)，显示 (Display)，执行器驱动 (Actuator Driver)，执行器监视器 (Actuator Monitor)
使用场合	控制系统

图 21-9 环境控制模式

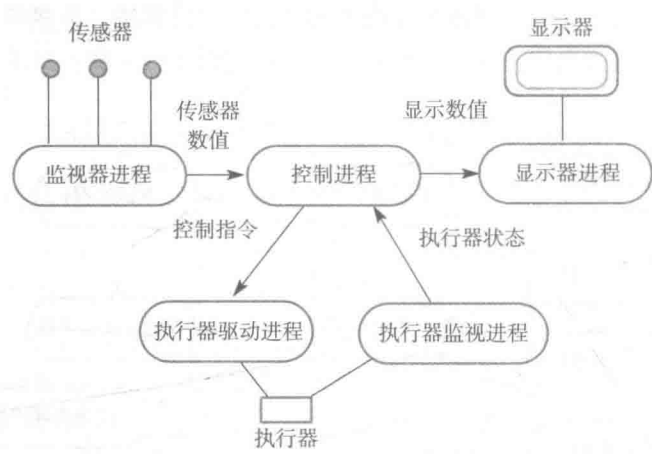


图 21-10 环境控制模式的进程结构

这个模式可以作为控制系统设计的基础。基本的做法是针对每一个执行器（或执行器类型）给出一个环境控制模式的实例。进一步的优化做法是减少进程的数量。例如，可以合并执行器监视进程和执行器控制进程，或者对几个执行器使用一个监视和控制进程。选择的优化取决于时序需求。可能监控传感器比发送控制信号更频繁，在此情形下，结合控制和监控进程就是不可行的了。在执行器控制和执行器监控进程之间也会存在直接反馈。这就允许我们通过执行器控制进程给出细粒度的控制决策。

图 21-11 示意了该模式是如何使用的。这是一个汽车刹车系统控制器的例子。设计的起始点是针对系统中的每种执行器类型给出一个模式实例。在这有 4 个执行器，每一个控制一个车轮上的制动。把单个传感器进程结合到一个监视所有车轮的车轮监视进程，该监视进程监视各个车轮上的传感器。传感器进程监视每一个车轮状态，判断车轮是转动还是抱死。一个单独的进程监视驾驶员踩刹车踏板的压力。

防滑制动系统包括防滑特征，在刹车时如果传感器提示车轮抱死则触发此特征，这意味着在路面和轮胎之间存在着不完全摩擦力；换句话说，车正在滑动。如果轮子被抱死，司机就不能控制轮子。为了对抗这一点，系统给此轮子的制动器发送一个快系列的开关信号，允许轮子转动并且接受控制。

车轮监视器监视每个车轮是否旋转。如果某个车轮打滑（未旋转），车轮监视器将通知分析进程。该进程随后发信号通知与打滑车轮关联的进程，启动防滑制动。

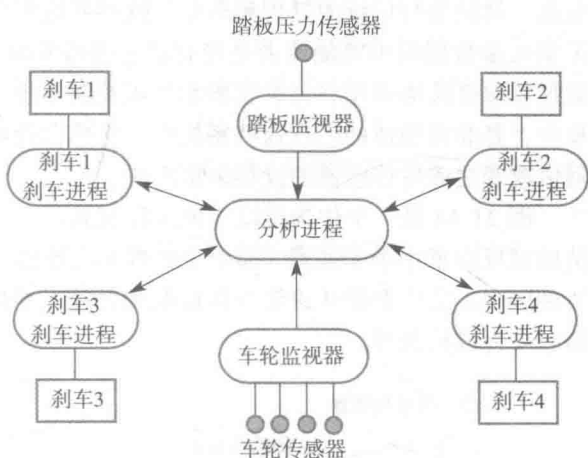


图 21-11 防滑制动系统的控制系统体系结构

21.2.3 处理管道模式

许多实时系统主要是从系统环境采集数据，然后将数据从原始数据表示转换成系统便于分析和处理的数字表示。实时系统也可以将数字数据转换成模拟数据，然后发送给它的环境。例如，软件收音机接收表示无线电传播的数值数据输入包，并且将这些数据转换成人们能听到的声音信号。

这些系统中，数据处理必须得到快速执行。否则，输入数据可能丢失并且输出符号可能由于基本信息的丢失而被破坏。处理管道模式使得这种快速处理成为可能。它把所需要的数据处理分解成一系列独立变换，且每个变换可以由一个独立的进程来执行。这对于使用多重处理器和多核处理器的系统是一个非常高效的体系结构。管道中每一个进程被分配到单独的处理器或者核上运行，因此处理步骤能并行执行。

图 21-12 是对数据管道模式的简洁描述，图 21-13 显示了这个模式的进程体系结构。注意，进程能够产生和消费信息。如同 21.1 节讨论的，这些进程利用同步缓冲区来交换信息。这就允许生产者和消费者进程以不同的速度工作而无数据损失。

名 称	处理管道
描述	处理管道使数据以序列形式从管道的一端移动到另一端。这些处理通常关联到一些同步缓冲区上，允许生产者和消费者进程以不同的速度运行。标志管道完成的是显示或数据存储，或者是管道连接到一个执行器
激励	输入来自环境或者其他进程的值
响应	输出值到环境中或者到一个共享缓冲区
进程	生产者（Producer），缓冲区（Buffer），消费者（Consumer）
使用场合	数据获取系统，多媒体系统

图 21-12 处理管道模式

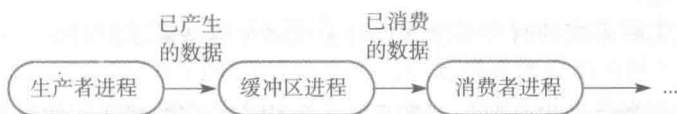


图 21-13 处理管道模式的进程结构

高速数据获取系统是使用管道处理模式的系统的一个例子。数据获取系统从传感器采集数据，然后进行后续的处理和分析。这些系统可以用于这样的情形，即传感器从系统环境中采集大量数据但不可能或者是没有必要进行实时处理，而是采集并存储以便之后用于分析。数据获取系统通常用在科学实验和过程控制系统中。过程控制系统中的物理过程，例如化学反应，是非常快速的。在这些系统中，传感器快速生成数据，数据获取系统需要确保在传感器值改变之前将传感器的读数采集出来。

图 21-14 是一个作为核反应堆中控制软件一部分的数据获取系统的简化模型。这是一个从监视反应堆中中子通量（即中子密度）的传感器采集数据的系统。传感器数据放置在一个缓冲区中，之后数据从该缓冲区提取和处理。平均通量水平显示在控制台显示器上，并且存储起来为以后处理。

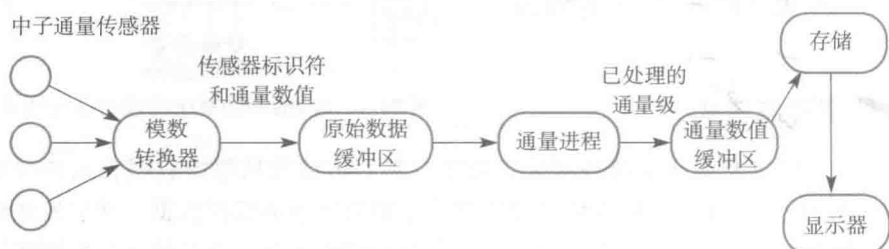


图 21-14 中子通量数据获取

21.3 时序分析

正如本章引言中所讨论的，实时系统的正确性不仅取决于其输出的正确性，而且还取决于产生这些输出的时间。因此，时序分析是嵌入式实时软件开发过程中的重要活动。在这样的分析中，你需要计算系统中每个过程必须执行的频率，以确保所有输入都得到处理，并且所有系统响应及时生成。时序分析的结果决定每个进程应该执行的频率，以及实时操作系统应该如何调度这些进程。

当系统必须混合处理周期性和非周期性的激励和响应时，实时系统的时序分析是相当困难的。因为非周期性激励是不可预测的，你必须假设这些激励发生并因此在任何特定时间要求服务的可能性。这些假设可能是不正确的，交付后的系统性能可能不足。Cooling 的书（Cooling 2003）讨论了考虑非周期事件的实时系统性能分析技术。

随着计算机处理速度越来越快，在许多系统中，仅使用周期性激励的设计已成为可能。当处理器缓慢时，就必须使用非周期性激励以确保关键事件在其截止时间之前被处理，而处理延迟通常给系统带来损失。例如，嵌入式系统中的电源供应故障可能意味着系统必须在非常短的时间（例如 50ms）内以受控的方式关闭所连接的设备。这可以被实现为“电源故障”中断。然而，这也可以使用频繁运行并检查电源的周期性进程来实现。只要进程调用之间的时间短，在电源故障引起损坏之前，仍然有时间执行系统的受控关闭。因此，本节只讨论周期性进程的时序问题。

当分析嵌入式实时系统的时序需求并设计系统满足这些需求的时候，需要考虑以下 3 个关键因素。

1. 最后时限，对激励做出必要处理和系统产生相应响应的时间。如果系统不能满足最后时限，那么，若它是硬实时系统，则这是系统失效；若它是软实时系统，这会导致系统服务

降级。

2. 频率，进程每秒必须执行的次数，以确保它总能满足其最后时限。

3. 执行时间，处理激励和产生响应所需的时间。执行时间并不总是相同的，这是由于代码的条件执行、等待其他进程的延迟等原因。因此，必须同时考虑进程的平均执行时间和该进程的最坏情况执行时间。最坏情况执行时间是进程执行所需的最大时间。在硬实时系统中，可能必须基于最坏情况执行时间做出假设，以确保不会错过最后时限。在软实时系统中，可以基于平均执行时间计算。

为了继续电源失效的示例，让我们计算将设备电源从主电源切换到备用电池的进程的最坏情况执行时间。图 21-15 显示了系统中的事件的时间线。

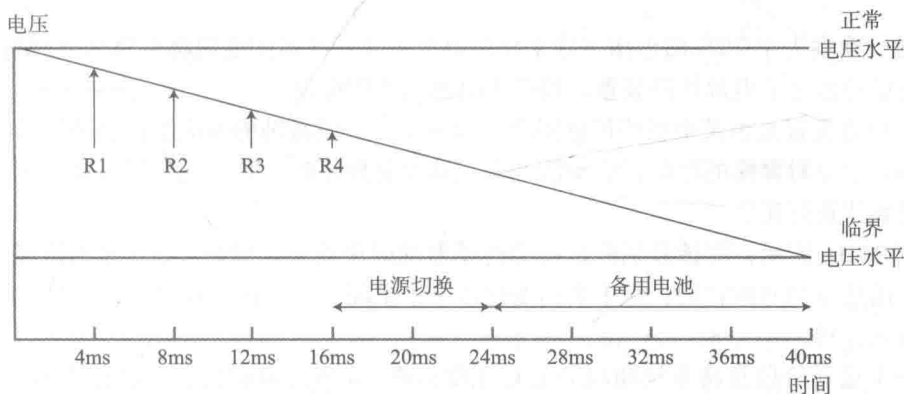


图 21-15 电源失效的时序分析

1. 假定主电源失效事件之后，所提供的电压下降到设备可能被损坏的水平需要 50ms。因此，必须在 50ms 内激活并运行备用电池。通常，会允许存在误差范围，因此，由于设备的物理变化，应该设置 40ms 作为最短时限。这意味着所有设备必须在 40ms 内在备用电池上运行。

2. 但是，备用电池系统无法立即激活。从启动备用电池到电池完全运行需要 16ms。这意味着可用于检测电源故障和启动备用电池系统的时间为 24ms。

3. 有一个进程被调度为每秒运行 250 次，即每 4ms 运行一次。该进程假定，如果在一次读取之间存在显著的电压下降并且持续 3 次读取，则存在电源故障。容许这个时间是为了不使临时波动而导致切换到电池备份系统。

4. 在上述时间线中，电源在一次读取完成后立即失效。因此，R1 读取是电源故障检查的开始读取。对于 R2 ~ R4 读取，电压继续下降，因此假定电源发生失效。这是最糟的可能情况，这种情况中传感器检查之后立即发生电源失效事件，因此从该事件起已经过去了 16ms。

5. 在此阶段，开始进行切换到备用电池的进程。由于备用电池需要 16ms 才能运行，此过程的最坏情况执行时间为 8ms，因此可以达到 40ms 的最后时限。

实时系统中时序分析的起始点是时序需求，它应该列出系统中每个所需响应的最后时限。图 21-16 展示了 21.2.1 节中讨论的办公大楼防盗警报系统可能的时序需求。为了简化这个例子，忽略由系统测试程序和外部信号产生的激励，以及在发生假报警时重置系统。这意味着系统仅处理两种类型的激励：

激励 / 响应	时序需求
声音警报	声音警报应在传感器产生警报后 0.5s 内打开
通信	对警察的呼叫应在传感器产生警报后 2s 内开始
门警报	每个门警报应每秒被轮询两次
电灯开关	灯应在传感器产生警报后半秒内打开
运动检测器	每个运动检测器应每秒被轮询两次
电源失效	到备用电源的切换必须在 50ms 的时限内完成
语音合成器	在传感器产生警报后 2s 内，应该有合成信息
窗警报	每个窗警报应每秒被轮询两次

图 21-16 防盗警报系统的时序需求

1. 通过观察大于 20% 的电压下降来检测电源失效。所需的响应将电路切换到备用电源，通过发送信号给电子电源切换装置，切换主电源到备用电池。

2. 入侵者警报是由某个系统传感器产生的激励。对该激励的响应是计算被激活传感器的房间号码，建立对警察的呼叫，启动语音合成器以管理呼叫，并且开启该区域中的可听到的入侵警报和建筑灯光。

如图 21-16 所示，应该分别列出每类传感器的时序约束，即使（如这里的情况）它们是相同的。通过分别考虑它们，为未来的变化留下了空间，并且使得计算每秒控制进程所需执行的次数更容易。

下一个设计阶段是将系统功能分配到并发进程。4 种类型的传感器必须周期性地轮询：电压传感器，门传感器，窗传感器和运动检测器；每种传感器都具有相关联的进程。通常，与这些传感器相关联的进程将非常快地执行，因为它们做的仅仅是检查传感器是否已经改变其状态（例如，从关闭到打开）。可以合理地假设，检查和评估一个传感器状态的执行时间是小于 1ms 的。

为了确保满足时序需求定义的最后时限，接下来必须决定相关进程运行的频率以及在进程的每个执行过程中应检查多少个传感器。在频率和执行时间之间存在明显的权衡。

1. 检测状态变化的最后时限为 0.25s，这意味着每个传感器必须每秒检查 4 次。如果在每个进程执行期间检查一个传感器，那么如果有 N 个特定类型的传感器，那么必须调度此进程每秒运行 $4N$ 次以确保在最后时限内检查所有传感器。

2. 如果在每个进程执行期间检查 4 个传感器，则执行时间增加到约 4ms，但是只需要运行该进程 N 次 /s 来满足时序需求。

在这种情况下，因为系统需求定义了两个或更多传感器被激活时的动作，最好的策略是分组检查传感器，其中分组依据是传感器的物理接近度。如果入侵者已经进入建筑物，则可能相邻的传感器是被激活的。

完成时序分析后，你可以使用有关执行频率及其预期执行时间的信息对进程模型进行注释（见图 21-17）。这里，对周期性进程用它们的频率注释，为响应激励而启动的进程注释为 R，测试进程是后台过程，注释为 B。该后台进程仅在处理器时间空闲时运行。通常，进程频率较低的系统设计是更简单的。执行时间表示进程所需的最坏情况执行时间。

设计进程的最后一步是设计一个调度系统，以确保一个进程总是被调度为满足其最后时限。只有知道所使用的实时操作系统（Operating System, OS）支持的调度方法，才能做到

这一点 (Burns 和 Wellings 2009)。实时操作系统中的调度程序在给定时间内向处理器分配进程。时间可以是固定的, 或者它可以根据进程的优先级而变化。

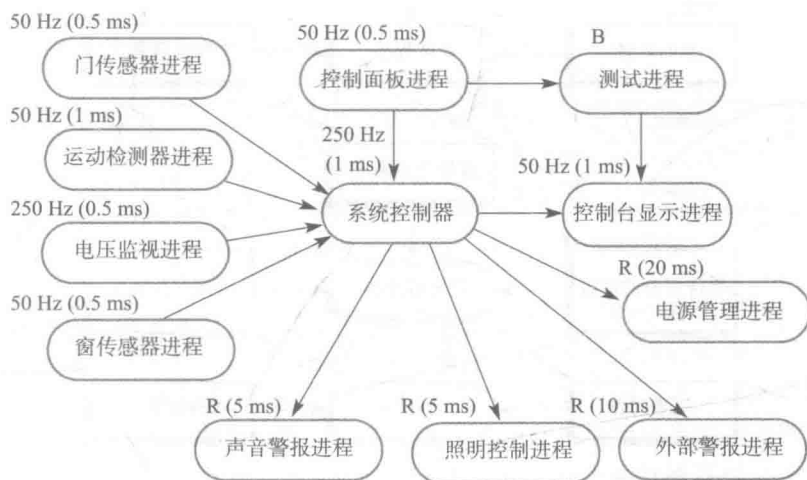


图 21-17 警报进程时序

在分配进程优先级时, 必须考虑每个进程的最后时限, 以使有较短时限的进程获得处理时间, 来满足它们的最后时限。例如, 防盗报警系统中的电压监视进程需要进行调度, 以便可以检测电压下降, 并在系统失效前切换到备用电源。因此, 这应当比检查传感器值的进程具有更高的优先级, 这是由于与后者的预期执行时间相比, 它们具有相当宽松的最后时限。

21.4 实时操作系统

大多数应用系统的执行平台是管理共享资源并提供诸如文件系统和运行时进程管理的特征的操作系统。然而, 常规操作系统中的大量功能占用了大量空间并且减慢了程序的操作。而且, 系统中的进程管理特征可能设计为不允许对进程调度进行细粒度控制。

由于这些原因, 诸如 Linux 和 Windows 等标准操作系统通常不用作实时系统的执行平台。非常简单的嵌入式系统可以被实现为“裸机”系统。系统提供自己的执行支持, 因此包括系统启动和关闭, 进程和资源管理以及进程调度。然而, 更通常的情形是, 嵌入式应用建立在实时操作系统 (Real-time Operating System, RTOS) 之上, 实时操作系统是提供实时系统所需特征的高效操作系统。实时操作系统的例子有 Windows Embedded Compact、VxWorks 和 RTLinux。

实时操作系统管理实时系统的进程和资源分配。它启动和停止进程, 以便激励可以被处理, 它还分配内存和处理器资源。实时操作系统的构件 (见图 21-18) 取决于正在开发的实时系统的大小和复杂性。对于除最简单系统之外的所有系统, 它们通常包括:

1. 实时时钟, 提供定期调度进程所需的信息。
2. 中断处理程序, 如果系统支持中断的话, 其管理非周期性服务请求。
3. 调度 (器) 程序, 负责检查可执行的进程, 并选择其中一个进程执行。
4. 资源管理器, 分配适当的内存和处理器资源给已经被调度执行的进程。
5. 分配器, 负责启动进程的执行。

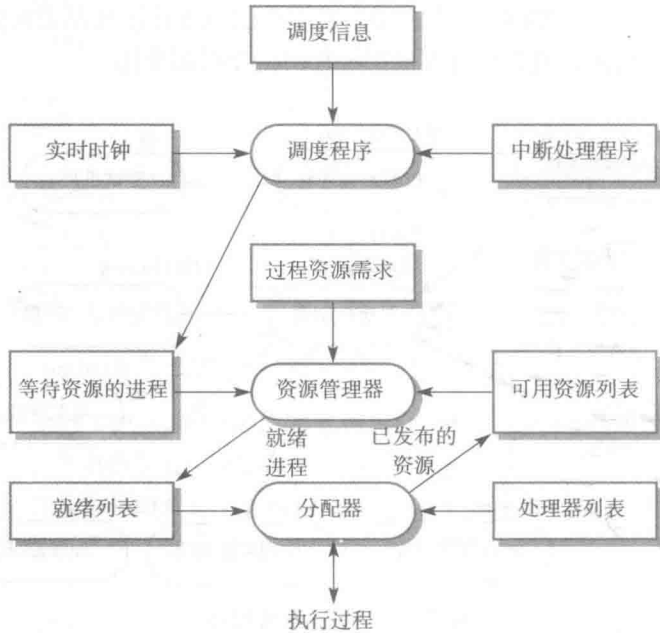


图 21-18 实时操作系统的构件

用于诸如过程控制或电信系统等大型系统等实时操作系统，可能有额外设施，即磁盘存储管理，检测和报告系统故障的故障管理设施，以及支持实时应用动态重配置的配置管理器。

21.4.1 进程管理

实时系统必须快速处理外部事件，并且在某些情况下，满足处理这些事件的最后时限。因此，事件处理进程必须被调度及时执行以检测事件，还必须给它们分配足够的处理器资源以满足其最后时限。实时操作系统中的进程管理器负责选择要执行的进程，分配处理器和内存资源，以及启动和停止处理器上的进程执行。

进程管理器必须管理具有不同优先级的进程。对于某些激励，例如与某些异常事件相关的激励，它们的处理必须在特定的时间期限内完成。如果有更关键的进程需要服务，则可以安全地延迟其他进程。因此，实时操作系统至少必须能够为系统进程管理以下两个优先等级。

1. 时钟等级。此优先等级分配给周期性进程。
2. 中断等级。这是最高优先等级。它被分配给需要非常快速响应的进程。这些进程之一将是实时时钟进程。如果系统不支持中断，则不需要此进程。

可以将进一步的优先等级分配给不需要满足实时时限的后台进程（例如自检验进程）。这些进程被调度在处理器容量可用时执行。

周期性进程必须在特定的时间间隔执行以用于数据获取和执行器控制。在大多数实时系统中，将会有几种类型的周期性进程。使用应用程序中指定的时序需求，实时操作系统安排周期性进程的执行，以便它们都能满足其最后时限。

操作系统对周期性进程管理采取的操作如图 21-19 所示。调度程序检查周期性进程列表并选择要执行的进程。选择取决于进程优先级、进程周期、预期执行时间和就绪进程的最后一

时限。有时,具有不同时限的两个进程应在同一时间点执行。在这种情况下,必须延迟其中一个进程。通常,系统将选择延迟具有最长时限的进程。

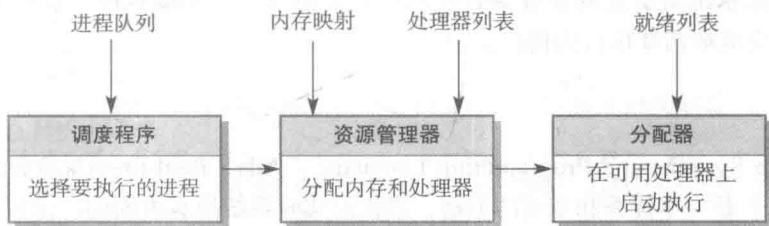


图 21-19 实时操作系统启动一个进程所需的操作

必须快速响应异步事件的进程可能是中断驱动的。计算机的中断机制导致控制转移到预先决定的内存位置。该位置包含一条跳转到简单快速中断服务例程的指令。服务例程使进一步的中断无效以避免自身中断。然后,它找出中断和启动的原因,并以高优先级启动处理引起中断的激励的进程。在一些高速数据获取系统中,中断处理程序将表示中断信号可用的数据保存在缓冲区中以便稍后处理。然后中断再次被激活,并且控制被返回给操作系统。

在任何一个时间,可以执行具有不同优先级的几个进程。进程调度程序实现决定进程执行顺序的系统调度策略。有两种常用的调度策略。

1. 非抢占式调度。在进程被调度执行之后,它运行到完成或直到它由于某种原因被阻塞,例如等待输入。如果存在具有不同优先级的进程,并且高优先级进程必须等待低优先级进程完成,则这可能导致问题。

2. 抢占式调度。如果较高优先级的进程需要服务,则可以停止当前执行进程的执行。较高优先级进程优先抢占较低优先级进程的执行,并被分配给处理器。

在这些策略中,已经开发了不同的调度算法。包括循环调度,其中每个进程轮流执行;速率单调调度,其中具有最短周期(最高频率)的进程被赋予优先级;最短期限优先调度,其中队列中具有最短时限的进程被调度(Burns and Wellings 2009)。

有关要执行进程的信息被传递给资源管理器。资源管理器分配内存,并且在多处理器系统中,还向该进程添加处理器。然后该进程被放在“就绪列表”中,这是一个包含准备好执行的进程的列表。当处理器完成执行进程并变得可用时,分配器被调用。它扫描就绪列表以查找能在可用处理器上执行的进程并开始执行它。

要点

- 嵌入式软件系统是硬件/软件系统的一部分,可对环境中的事件做出反应。软件被“嵌入”在硬件中。嵌入式系统通常是实时系统。
- 实时系统是必须实时响应事件的软件系统。系统正确性不仅取决于其产生的结果,而且取决于产生这些结果的时间。
- 实时系统通常被实现为一组对激励做出反应以产生响应的通信进程。
- 状态模型是嵌入式实时系统的重要设计表现。它们用于显示当事件触发系统中状态变化时系统如何对其环境做出反应。
- 在不同类型的嵌入式系统中,可以观察到几种标准模式。这包括用于针对不良事件监视系统环境的模式,用于执行器控制的模式以及数据处理模式。

- 实时系统的设计者必须进行时序分析，这是由对激励处理和响应的最后期限驱动的。它们必须决定系统中每个进程应该运行的频率以及进程的预期和最坏情况执行时间。
- 实时操作系统负责过程和资源管理。它总是包含一个调度程序，后者是负责决定哪个进程应该被调度执行的构件。

阅读推荐

《Real-time Systems and Programming Language : Ada, Real-time Java and C/Real-time POSIX(4rd ed.)》是一本优秀和全面的书籍，提供对实时系统所有方面的广泛覆盖。(A. Burns and A. Wells, Addison-Wesley, 2009)

《Trends in Embedded Software Engineering》这篇论文提出模型驱动开发(如本书第5章所讨论的)将成为嵌入式系统开发的重要方法。这是嵌入式系统的一个特殊问题的一部分。其他文章，如Ebert和Jones的这篇文章，阅读也很受益。(IEEE Software, 26(3), May-June 2009) <http://dx.doi.org/10.1109/MS.2009.80>

《Real-time systems : Design Principles for Distributed Embedded Applications, 2nd ed.》是一本关于现代实时系统的综合教科书，所涉及的实时系统可以是分布式和移动系统。作者关注硬实时系统，并涵盖了如互联网连接和电源管理这样重要的主题。(H. Kopetz, Springer, 2013)

网站

本章的PPT: <http://software-engineering-book.com/slides/chap21/>

支持视频的链接: <http://software-engineering-book.com/videos/systems-engineering/>

练习

- 21.1 用例子解释为什么实时系统通常需要使用并发进程来实现。
- 21.2 识别在家用冰箱或家用洗衣机中的嵌入式系统的可能的激励和响应。
- 21.3 使用基于状态的方法建模，如在21.1.1节中讨论过的，为有线电话中的语音邮件系统的嵌入式软件系统的控制建模。应该在LED上显示记录的消息的数量，应该允许用户拨打并接听记录的消息。
- 21.4 为什么面向对象软件开发方法可能不适合实时系统的开发？
- 21.5 给出如何使用环境控制模式作为温室温度控制系统的设计基础。温度应该在 $10 \sim 30^{\circ}\text{C}$ 。如果温度低于 10°C ，就应该将供暖系统打开；如果温度超过 30°C ，窗户就应该自动打开。
- 21.6 设计一个环境监控系统的进程的体系结构，该系统从一组安装在城市周围的空气质量传感器收集数据。5000个传感器被分成100组。每个传感器每秒要被查询4次。当某一区域超过30%的传感器指示空气质量低于一个可接受的水平时，局部警告灯就被打开。所有传感器将数据返回给中央处理计算机，这台计算机每15min产生一次该城市的空气质量报告。
- 21.7 一个火车保护系统自动地在达到某个路段限速或某一路段当前有红灯显示(该路段不得进入)时对火车实行制动，细节内容如图21-20所示。写出火车上控制系统要处理的激励以及相关的响应。

火车保护系统的需求

- 该系统从轨道侧发射机获取关于该轨道段速度限制的信息，发射机持续广播该段标识符及其速度限制。相同的发射机还广播关于控制该轨道段信号的状态的信息。广播轨道段和信号信息所需的时间是 50ms。
- 当列车在距离发射机 10m 范围内时，可以从轨道发射机接收信息。
- 最高列车速度为 180km/h。
- 列车上的传感器提供有关当前列车的速度（每 250 ms 更新一次）和列车制动状态（每 100 ms 更新一次）的信息。
- 如果列车速度超过当前轨道段限速多于 5km/h，则在驾驶室中发出警告。如果列车速度超过当前轨道段限速多于 10km/h，则列车制动自动启动直到速度下降到轨道段限速范围。当检测到过高的列车速度时，列车制动应在 100ms 内启动。
- 如果列车进入信号灯是红灯的轨道段，列车保护系统启动列车制动，并将列车速度降至零。列车制动应在接收到红灯信号的 100ms 内启动。
- 系统会不断更新驾驶室内的状态显示。

图 21-20 火车保护系统的需求

21.8 为练习 21.7 中的系统绘制出一个可能的进程体系结构。

21.9 在火车保护系统中，要有一个用来收集从轨道旁发射机中发送的信息的一个周期性进程，这个周期性进程需要什么样的调度频度才能确保系统能收集到从发射机发送来的信息？解释如何实现你的答案。

21.10 为什么通用目的操作系统例如 Linux 或 Windows 不适合作为实时系统平台？根据你使用通用目的系统的经验来回答此问题。

参考文献

- Berry, G. 1989. "Real-Time Programming: Special-Purpose or General-Purpose Languages." In *Information Processing*, edited by G. Ritter, 89:11–17. Amsterdam: Elsevier Science Publishers.
- Bruno, E. J., and G. Bollella. 2009. *Real-Time Java Programming: With Java RTS*. Boston: Prentice-Hall.
- Burns, A., and A. Wellings. 2009. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Boston: Addison-Wesley.
- Cooling, J. 2003. *Software Engineering for Real-Time Systems*. Harlow, UK: Addison-Wesley.
- Douglass, B. P. 1999. *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd ed.* Boston: Addison-Wesley.
- . 2002. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley.
- Ebert, C., and C. Jones. 2009. "Embedded Software: Facts, Figures and Future." *IEEE Computer* 26 (3): 42–52. doi:10.1109/MC.2009.118.
- Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.
- . 1988. "On Visual Formalisms." *Comm. ACM* 31 (5): 514–530. doi:10.1145/42411.42414.
- Lee, E. A. 2002. "Embedded Software." In *Advances in Computers*, edited by M. Zelkowitz. Vol. 56. London: Academic Press.
- Silberschaltz, A., P. B. Galvin, and G. Gagne. 2013. *Operating System Concepts, 9th ed.* New York: John Wiley & Sons.
- Stallings, W. 2014. *Operating Systems: Internals and Design Principles, 8th ed.* Boston: Prentice-Hall.

软件管理

有人认为软件工程与其他类型的编程的关键区别在于软件工程是一个受管理的过程。这样说的意思是软件开发是在一个组织内进行的，受到一系列进度、预算以及其他组织约束。本书的这一部分介绍了一系列的管理话题，并关注技术管理问题，而不是“更软”的管理问题（例如人员管理）或者更具战略性的企业系统管理问题。

第 22 章和第 23 章关注基本的项目管理活动、计划、风险管理和人员管理。第 22 章介绍软件项目管理，22.1 节主要关注风险管理。在风险管理中，管理者要确定哪些地方可能出问题并规划如何应对这些问题。这一章还会介绍人员管理和团队协作。

第 23 章介绍项目计划和估算。本章将条形图作为基本的项目计划工具，并解释为什么敏捷方法虽然很成功，但计划驱动的开发仍将继续作为一个重要的开发方法存在。此外，本章还将讨论影响系统定价的一些问题以及软件成本估算的技术，并使用 COMOCO 成本模型族来描述算法成本建模，并分析此方法的优点和缺点。

第 24 章根据大型项目中的实践介绍软件质量管理的基础。质量管理是用于确保和改善软件质量的过程和技术。这一章讨论了质量管理标准的重要性，以及如何在质量保障过程中使用评审和审查技术。这一章的最后一节介绍了软件度量，讨论了在质量管理中使用度量和软件数据分析的好处和问题。

最后，第 25 章讨论配置管理，这是一个对所有大型系统都很关键的问题。然而，对于只关注个人软件开发的学生而言，对配置管理的重要性却不总是很清楚。所以这一章描述了配置管理的各个方面，包括版本管理、系统构建、变更管理以及发布管理。这一章解释了为什么持续集成或每日系统构建很重要。本书这一版的一个重要更新是增加了分布式版本管理系统的材料，例如 Git，这些系统已经在软件工程中得到越来越广泛的应用，尤其是在分布式团队中。

项目管理

目标

本章的目的是介绍软件项目管理和两项重要的管理活动，即风险管理和人员管理。阅读完本章后，你将：

- 了解软件项目管理者的主要任务；
- 了解风险管理的概念以及在软件项目中可能出现的一些风险；
- 理解影响工作积极性的因素，以及这些因素对于软件项目管理者的意义；
- 理解影响团队协作的主要问题，例如团队的构成、团队的组织和团队的沟通。

软件项目管理是软件工程的一个重要组成部分。需要对软件项目进行管理，这是因为专业的软件工程总是要受预算和工程进度的制约。软件项目管理者的任务是确保软件项目满足和服从这些约束，并确保交付高质量的软件产品。好的管理不能确保项目成功，但是不好的管理注定会造成项目失败：软件可能会延迟交付，成本超出预期，或者无法满足客户的期望。

很显然项目管理成功的标准对于不同的项目是不同的，但是对于大多数项目来说，最重要的目标是：

- 按照所约定的时间将软件交付给客户；
- 将全部成本控制在预算之内；
- 所交付的软件满足客户的期望；
- 保持一个有凝聚力并且运作良好的开发团队。

这些目标并不是软件工程所独有的，而是所有工程项目的目标。然而，软件工程管理与其他工程管理相比，在很多方面有显著的区别，这使得软件工程管理具有相当大的难度。以下列出了软件工程管理的一些不同之处。

1. 软件产品是无形的。一个造船或土木工程项目的管理者能够看见正在制造的产品。如果进度跟不上，那么就能从产品的现状中直接看出来：整个结构中的某些部分很明显是未完工的。而软件产品是无形的，看不见摸不着。软件项目管理者不能依靠查看正在开发的产品来了解进展情况。在一定程度上，他们只能依靠其他人提供信息来掌握工作进度。

2. 大型软件项目常常是“一次性”的项目。每个大型软件项目都是独特的，因为每一个软件开发所处的环境在某些方面都各不相同。每个管理者纵然有通过计划降低不确定性的经验，也很难预见将出现的问题。此外，计算机和通信技术飞速发展，早先的经验很快变得过时了，其中的经验教训不能在新的项目中发挥作用。

3. 软件开发过程是可变的并且与特定组织相关。对于某些系统，例如桥梁和建筑等类型的工程过程我们已经了如指掌。然而，软件开发过程对于不同的组织而言是相当不同的。虽然我们在过程标准化和过程优化方面已经取得了很大的进步，但是不同的公司经常使用很不一样的软件开发过程。我们还不能确切地预见某一软件过程何时有可能出现问题。特别当这个软件项目是某个更大的系统工程项目的一部分或者当开发一个全新的软件时尤为明显。

基于以上这些原因,一些软件项目延期、超出预算、进度慢也就不足为奇了。软件系统往往是全新的、非常复杂的而且技术上也有所创新。其他复杂和具有创新性的工程项目(例如新的运输系统)也经常会有存在进度问题和成本超支问题。一旦出现困难,软件项目的按期、按预算完成就会变得几乎不太可能。

软件管理者的工作内容没有一定的标准。开发组织和开发的软件决定着管理工作的内容。下面列出一些影响软件项目管理的最重要的因素。

1. 公司规模。小公司能够在使用非正式的管理方式和团队沟通方式的情况下继续运行下去,也不需要正式的政策和管理结构。与更大的组织相比,小公司管理开销更少。对较大的组织而言,管理层次结构、正式的报告和预算以及审批流程这些都是必须遵循的。

2. 软件客户。如果软件的客户是内部客户(软件产品开发就是这种情况),那么客户沟通可以是非正式的,也没必要适应客户的工作方式。如果软件是为外部客户开发的,那么必须通过更正式的沟通方式达成一致。如果客户是政府组织,那么软件公司必须遵循政府组织的政策和规程进行操作,这一过程很可能会有些官僚主义。

3. 软件规模。小系统由小团队就能够开发出来,这样的小团队中的成员可以在同一个房间内一起来讨论进展和其他管理问题。大系统通常需要多个开发团队来开发,这些团队可能分散在不同的地方和不同的公司。因此,项目管理者必须协调这些团队的活动,并安排他们互相之间的交流。

4. 软件类型。一方面,如果正在开发的软件是一个消费性产品,那么项目管理决策的正式记录就不是必需的。另一方面,如果正在开发的是一个安全性关键系统,那么所有的项目管理决策都应该被记录而且所有决策都应该是合理的,因为它们可能影响系统的安全性。

5. 组织文化。一些组织的文化基于支持和鼓励个体,而其他一些组织的文化关注小组。大组织通常都是官僚主义的。一些组织的文化是敢于冒险的,而其他一些组织的文化是不愿承担风险的。

6. 软件开发过程。敏捷过程通常会尝试采用“轻量级”管理。更加正式的过程需要管理监控来确保开发团队遵循已定义好的过程。

这些因素意味着在不同组织中的项目管理者可能采用完全不同的方式工作。然而,很多基础项目管理活动对于所有组织而言是通用的。

1. 项目计划。项目管理者负责计划、评估以及调度项目开发,并给人员分配任务。他们监督工作确保工作是按要求的标准进行,他们还监督开发过程,检查开发是否按时进行以及是否在预算之内。

2. 风险管理。项目管理者必须评估并且监测某些可能影响项目的风险,并在问题出现时立刻采取行动。

3. 人员管理。项目管理者负责管理一个团队。他们必须为他们的团队挑选人员,并且建立能够使团队高效运作的工作方式。

4. 工作报告。项目管理者通常有责任向客户和开发此软件的公司的经理汇报项目进展情况。他们必须能够进行各个层次的交流,从详尽的技术细节到概要的管理总结。项目管理者必须能从详细的项目报告中抽出重要的信息,写成简明、一致的文档,并在项目评审时陈述出来。

5. 编写建议书。软件项目的第一个阶段通常都会编写一个建议书以赢得某项工作的合同。建议书描述了项目的目的以及如何实施。其中通常会包含成本和进度估计,并说明为什么项目合同应该被给予某个特定的组织或团队。编写建议书是一个关键性的任务,因为许多

软件公司的生存都依赖于有足够多的项目建议书被接受并获得项目合同。

项目计划是一项重要的主题，将在第 23 章讨论。本章主要介绍风险管理和人员管理。

22.1 风险管理

能预见可能影响正开发的软件的项目进度或产品质量的风险，并采取行动避免这些风险，是项目管理者的一项重要任务（Hall 1998；Ould 1999）。可以把风险看作是一些可能实际发生的不利因素。风险可能危及整个项目、正在开发的软件或者开发组织。

如 22.1.1 节中所说，风险可以根据风险的类型（技术风险、组织风险等）来分类。另一个补充的分类法是根据风险的影响对风险进行分类。

1. 项目风险是影响项目进度或项目资源的风险。例如失去一位经验丰富的系统架构师。寻找一位经验和技能满足要求的替代者可能需要很长一段时间，而这带来的直接后果就是软件的设计需要更长的时间才能完成。

2. 产品风险是影响开发中软件的质量或性能的风险。例如，购买的构件不能达到预期目标。这可能会影响整个系统的性能，导致整个系统运行速度下降。

3. 业务风险是影响软件开发组织或软件产品购买组织的风险。例如，竞争对手推出了新的产品。竞争产品的推出可能预示着，先前做出的关于现有软件产品的销售情况可能过于乐观。

当然，这些风险分类之间会有重叠。例如，经验丰富的程序设计师离职，由此表现出来的的是一个项目风险，因为软件交付进度将受到影响。新来的项目成员不可避免要花一些时间来熟悉这个项目，并不能立刻高效地投入工作。因此，系统的交付可能延期。失去一个团队成员也可以是一个产品风险，因为继任者可能不如前者有经验，因而可能犯编程错误。最后，失去一个团队成员所带来的风险也可能是业务风险，因为有经验的程序员的声望是取得新的合同的关键。

对于大项目，项目管理者应该将风险分析的结果以及相应的后果，包括对项目的风险后果，对产品的风险后果以及对业务的风险后果，写到风险记录中。有效的风险管理能够使项目管理者对未来出现的问题处理自如，并保证这些风险不会导致不可接受的预算和进度偏差。对于小项目，正式的风险记录可能不是必需的，但项目管理者应该对它们有所了解。

项目的风险类型取决于项目本身的特点和软件开发的组织环境。然而有许多共性的风险不依赖于所开发的软件的种类，即这些风险可能发生在任何项目中。图 22-1 给出了其中的一部分共性风险。

风 险	影 响	描 述
人员流动	项目	有经验的项目成员可能会在项目完成之前离开
管理层变更	项目	管理层结构可能会发生变化，不同的管理层考虑、关注的事情会不同
硬件不可用	项目	项目所需的基础硬件可能没有按期交付
需求变更	项目和产品	软件需求与预期的相比，可能会有许多变化
规格说明延迟	项目和产品	重要接口的规格说明未如期提供
低估了系统规模	项目和产品	过低估计了系统的规模
软件工具性能较差	产品	支持项目的软件工具达不到预期的性能
技术变更	业务	系统构造所依赖的基础技术被新技术取代
产品竞争	业务	系统还未完成，其他有竞争力的产品就已经上市了

图 22-1 常见的项目、产品和业务风险例子

风险管理对软件项目而言尤为重要,因为软件开发存在固有的不确定性。这些不确定性产生于:宽泛定义的需求(需求随客户要求的改变而改变),对软件开发所需时间和资源估算的困难,以及项目组成员技术上的差异性。项目管理者应该预见风险,了解这些风险对项目、产品和业务的冲击,并采取措施规避这些风险。可以制定应急计划,这样一旦风险来临,才有可能采取快速防御行动。

图 22-2 是说明风险管理过程的概要图。该过程包括以下几个阶段。

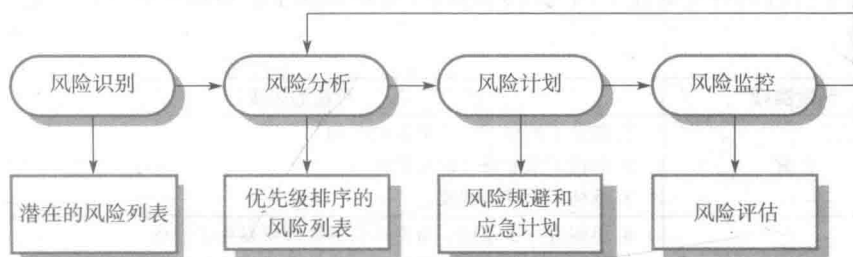


图 22-2 风险管理过程

1. 风险识别,识别可能的项目风险、产品风险和业务风险。
2. 风险分析,评估这些风险出现的可能性及其后果。
3. 风险计划,制订计划说明如何规避风险或最小化风险对项目的影响。
4. 风险监控,定期对风险和缓解风险的计划进行评估,并随着有关风险信息增多及时修正缓解风险的计划。

对于大型项目,风险管理过程的结果应该记录在风险管理计划中,具体包括:对项目所面临的风险的讨论,对这些风险的分析,以及当风险可能成为一个现实问题时管理这些风险的建议。

风险管理过程也是一个贯穿项目全过程、迭代进行的过程,从最初的计划制订开始,项目就处于被监控状态以检测可能出现的风险。随着有关风险的信息增多,需要重新进行分析,重新确定风险的优先级。对风险规避和应急计划要进行修正。

敏捷开发中的风险管理更加非正式。虽然可能没有正式的文档记录,但相同的基础活动仍要遵循,并且风险也已经如前面讨论过。敏捷开发减少了一些风险,例如需求变更造成的风险。然而敏捷开发也有缺点。因为敏捷开发依赖于人,所以人员流动对项目、产品和业务都有较大的影响。由于缺少正式文档以及依赖于非正式的沟通,因此,如果关键人员离开了项目,那么保持项目的连续性和积极性就变得非常困难。

22.1.1 风险识别

风险识别是风险管理的第一阶段。这一阶段主要是发现可能对软件工程过程、正在开发的软件或者开发组织产生重大威胁的风险。风险识别可以通过项目组对可能的风险的集体讨论完成,或者管理者凭借之前项目中出错的经验识别风险。

作为风险识别的起始点,可以使用一个关于不同类型风险的检查表。该风险检查表中可能会包含如下这 6 种风险。

1. 估算风险,源于管理层对构建系统所需资源进行的估算。
2. 组织风险,源于软件开发所处的组织环境。

- 3. 人员风险，与软件开发团队的成员相关。
- 4. 需求风险，源于客户需求的变更和需求变更的处理过程。
- 5. 技术风险，源于开发系统的软件技术或硬件技术。
- 6. 工具风险，源于软件工具和其他用于系统开发的支持软件。

图 22-3 对上述每一种风险都给出了可能的风险实例。风险识别过程的结果应该是列出一长串可能发生的风险，这些风险可能影响到软件产品、过程或业务。紧接着，需要削减风险列表到便于管理的程度为止。因为所列的风险实在太多了，所以实际工作中是不可能追踪所有的风险。

风险类型	可能的风险
估算	1. 低估了软件开发所需要的时间。 2. 低估了需要修复的缺陷比率。 3. 低估了软件的规模。
组织	4. 组织进行了重构，负责项目的管理层发生了变化。 5. 组织的财政状况出现问题，导致项目预算被削减。
人员	6. 招聘不到符合项目技术要求的人员。 7. 在项目的重要时间点上，关键的项目组成员生病并且无法发挥作用。 8. 人员所需的培训跟不上。
需求	9. 需求发生变更，主体设计要返工。 10. 客户不了解需求变更对项目造成的影响。
技术	11. 系统所使用的数据库每秒处理事务数量无法达到预期。 12. 可复用的软件构件中的故障必须在构件复用前被修复。
工具	13. 软件代码生成工具生成的代码效率低。 14. 软件工具无法以一种集成化的方式一起使用。

图 22-3 不同类型风险的例子

22.1.2 风险分析

在风险分析过程中，必须考虑每个已识别的风险并针对该风险的可能性和严重程度做出判断。风险分析不存在捷径。必须依赖于主观判断以及来自于过去的项目和其中所发生的问题的经验。不可能对每个风险的可能性和严重程度做出精确、数字化的评估；而是应该对风险的可能性和严重程度给出一个等级的判断。

- 1. 风险出现的可能性程度可以是微乎其微、低、中、高和非常高；
- 2. 风险的严重程度可以是灾难性的（严重威胁项目的存活）、严重的（可能引起很大的延迟）、可以容忍的（可能引起小的延迟）和可以忽略的。

风险分析过程结束后，应该根据风险严重程度按顺序制成表格。图 22-4 是对图 22-3 中已识别的风险进行分析，得出结果后做成的表格。很显然，这里对可能性和严重性的评估有些武断。为了做好这项评估，项目经理需要根据来自项目、过程、开发团队和组织状况等有关更详细的信息进行评估。

当然，随着有关风险的可用信息的增多和风险管理计划的实施，一项风险出现的可能性和对这一风险的影响后果的评估都可能改变。因此，这个表格必须在风险过程的每个迭代期间得到更新。

风险一经分析和排序，下一步就该判断哪些风险是最重要的，这是在项目期间必须考虑的。做出以上判断必须综合考虑风险出现的可能性大小和该风险的影响后果。一般而言，所

有灾难性的风险都是必须考虑的，所有出现的可能性超过中等、影响严重的风险同样要给予认真对待。

风 险	可能性	严重程度
组织的财政状况出现问题，导致项目预算被削减（5）	低	灾难性的
招聘不到符合项目技术要求的人员（6）	高	灾难性的
在项目的重要时间点上，关键的项目组成员生病导致无法发挥作用（7）	中	严重的
可复用的软件构件中的故障必须在构件复用前被修复（12）	中	严重的
需求发生变更，主体设计要返工（9）	中	严重的
组织进行了重构，负责项目的管理层发生了变化（4）	高	严重的
系统所使用的数据库每秒处理事务数量无法达到预期（11）	中	严重的
低估了软件开发所需要的时间（1）	高	严重的
软件工具无法集成（14）	高	可容忍的
客户不了解需求变更对项目造成的影响（10）	中	可容忍的
人员所需的培训跟不上（8）	中	可容忍的
低估了需要修复的缺陷比率（2）	中	可容忍的
低估了软件的规模（3）	高	可容忍的
软件代码生成工具生成的代码效率低（13）	中	可以忽略

图 22-4 风险类型和例子

Boehm（Boehm 1988）曾建议识别和监控“前 10 位”风险，笔者认为必须根据项目自身的情况确定要进行监控的风险的数量。可能是 5 个，也可能是 15 个。在图 22-4 已识别的风险中，考虑具有灾难性或严重后果的总共 8 种风险已经足够了（见图 22-5）。

22.1.3 风险计划

风险计划过程通过制订策略来对威胁项目的关键风险进行管理。对于每个风险而言，必须思考一旦某个风险发生时所需要采取的行动，使其对项目的干扰最小化。同时，应该考虑在监控项目时需要收集哪些信息，以便在出现的问题变严重之前检测出问题所在。

在风险计划中，必须会问“如果 - 应该怎么办”问题。所考虑的问题应该既包括单个风险又包括多个风险的混合以及影响这些风险的外部因素。下面列出了一些可能会问的问题。

- 1. 如果多个工程师同时生病了，应该怎么办？
- 2. 如果经济衰退导致项目的预算削减了 20%，应该怎么办？
- 3. 如果开源软件的性能不够好并且唯一了解这个软件的专家离职了，应该怎么办？
- 4. 如果提供并维护软件构件的公司破产了，应该怎么办？
- 5. 如果客户没有按预期提交修改后的需求，应该怎么办？

基于这些“如果 - 应该怎么办”问题，可以制订相应的策略来管理风险。

图 22-5 给出了处理图 22-4 中重大风险（严重的或灾难性的）的可能的策略。这些策略分为以下 3 类。

- 1. 规避策略。采用这些策略就会降低风险出现的可能性。图 22-5 中处理有缺陷的构件策略就是一个例子。
- 2. 最小化策略。采用这些策略就会减小风险的影响。图 22-5 中解决职员生病问题的策略就属于这一种。

风 险	策 略
组织的财务问题	拟一份简短的报告，提交高级管理层，说明这个项目将对业务目标有重大贡献，陈述理由，说明削减预算并不符合成本收益
人员招聘问题	告诉客户项目潜在的困难和延迟的可能性，研究购买构件的可能性
人员生病问题	重新对团队进行组织，使更多工作有所重叠，这样，团队成员可以了解他人的工作
有缺陷的构件	用买进的可靠性稳定的构件更换有潜在缺陷的构件
需求变更	建立可追踪信息来评估需求变更带来的影响；在设计中尽量做到信息隐藏
组织重构	拟一份简短的报告，提交给高级管理层，说明这个项目对于业务目标有非常重要的贡献
数据库的性能	研究购买更高性能数据库的可能性
低估开发时间	研究买入构件的可能性；研究使用自动化代码生成的可能性

图 22-5 风险管理策略

3. 应急计划。采用这些策略意味着就算最坏的事情发生，你也是有备而来，有适当的策略加以应对。图 22-5 中应对组织的财务问题的策略就属于这一类。

可以看到此处与在关键性系统中为确保可靠性、信息安全性和安全性所采取策略的相似之处——都必须规避、容忍失效以及从失效中恢复。显然，最好是使用规避风险的策略。如果这办不到的话，就采取降低那些会导致严重后果的风险的发生概率的策略（最小化策略）。最后，必须有成熟的应急策略以应对这些可能出现的风险，以此降低在项目或产品上的总的风险影响。

22.1.4 风险监控

风险监控就是检查之前对产品、过程以及业务风险的假设是否改变的过程。必须要对每一个识别的风险定期进行评估，从而确定风险出现的可能性是变大了还是变小了，风险的影响后果是否有所改变。为了达到这个目的，必须关注能提供有关风险可能性及其影响后果信息的其他因素，例如需求变更的数量。它能给出风险概率和风险影响的线索。显然，具体有哪些因素取决于风险的类型。图 22-6 举出上述因素的一些例子，可能会对评估这些风险类型有帮助。

风险类型	潜在的监测指标
估算	跟不上双方协商的进度；无法完全消除所报告的缺陷
组织	组织内的流言蜚语；高级管理层缺少动作
人员	人员士气低迷；团队成员的关系不好；人员流动率高
需求	很多需求变更请求；客户抱怨
技术	硬件或支持软件延迟交付；报告了很多技术问题
工具	团队成员不愿使用工具；抱怨软件工具；要求更快的计算机 / 更大的内存等。

图 22-6 风险监测指标

风险监控应该是一个持续不断的过程，在每一次对风险管理进行评审时，每一个重大风险都应该单独评审并在会上进行讨论。项目管理者应该判断风险出现的可能性是变大还是变小了，以及风险的严重性和后果是否也发生了变化。

22.2 人员管理

对于一个软件组织来说,人是这个组织中最重要资产。招募和留住好的员工的代价是很昂贵的,而项目中的工程师能否尽可能高效率地工作主要取决于软件管理者。成功的公司和经济实体尊重它们的员工,并且在对他们的任务分配上体现他们的技能和经验的价值,以此达到上述目标。

软件项目的管理者对影响软件开发工作的技术问题的理解是很重要的。然而不幸的是,好的软件工程师并不一定是好的人力资源管理者。软件工程师通常拥有娴熟的技术技巧,但是却缺乏能够激励和领导项目开发团队的软技术。作为项目管理者,应该意识到人员管理的潜在问题,并且应该尝试不断提高自身在人员管理方面的技巧。

有4个关键要素影响着管理人员与他所管理的团队成员之间的关系。

1. 一致性。对项目组的每个人应该同等对待。虽然人们不会期望每个人的报酬都相同,但是不应该让项目组成员感到自己对组织的贡献被低估了。

2. 尊重。不同的人有着不同的技能,管理者应尊重这些差异。团队的所有成员都应有机会做出贡献。当然在某些情况下你会发现某个人根本不适合在这个团队,所以也不能够继续下去,但重要的是不要匆忙得出这样的结论。

3. 包容。当人们觉得其他人能够倾听并能采纳自己的建议时,他们能更有效地工作。很重要的一点是创造一种工作环境,在这种工作环境下,所有的见解,甚至是最底层员工的意见都会得到考虑。

4. 诚实。作为一个管理者,必须对项目组中好的情况和不好的情况保持诚实的态度。应当诚实地对待自己的技术知识水平,并且在必要的时候乐于服从更博学的员工。如果掩盖存在的问题和自己的无知,最终将被发现并且失去团队的尊敬。

实用的人员管理方法必定是基于以往的经验,所以在这一节和下一节讲团队合作的内容,目的是使管理者意识到那些他们要解决的最重要的问题。

22.2.1 激励人员

项目管理者需要激励一起工作的员工,使他们能尽心尽力。激励意味着在协调工作和提供良好工作环境方面能使员工尽可能更有效率地工作。如果员工没有被激励起来,他们将失去对正在从事的工作的兴趣,工作进度会很缓慢,会更容易犯错误,并且不能对团队或组织长远的目标有所贡献。

为了达到激励员工的目的,需要理解什么才能激励员工。马斯洛(Maslow 1954)建议通过满足人的需求来激励他们,这些需求被划分成一系列层次,如图22-7所示。这里较低的层次表示基本的需求,如对食物、睡眠等的需求,还有在某一环境下对安全的需求。社会需求是指对某一社会群体的认同感;受尊重的需求是指得到其他人的尊重;自我实现的需求是指个人发展。人类首先要满足低层次的需求,如饥饿,然后是更抽象更高层次的需求。

软件开发组织中的员工根本没有饥渴问题,一般也不会感受到环境对身体的威胁。因此,确保社会需



图 22-7 人的需求层次

求、受尊重需求和自我实现需求的满足，从管理的观点来看是最有意义的。

1. 满足员工的社会需求，为项目组成员提供与同事交谈的时间和场所。一些软件公司（例如谷歌）就在办公室里提供了社交空间，使员工可以聚集在一起。当所有开发团队成员在一个地方工作时，这种需求相对较容易满足，但越来越多的团队成员并不在同一个建筑物中工作，甚至不在同一个城市或一个国家工作。他们可能同时在为多家公司服务，还可能绝大多数时间里是在家工作。社交网络系统和电话会议可以使交流变得更便捷。通常，电子系统只有在人们互相认识的情况下使用才最有效。所以，在项目的初期应当安排定期的面对面的会议，以使人们能够直接与团队的其他成员进行交流。通过这种直接交流，人们成为这个团队的一部分，并理解团队的目标和重点。

2. 为了满足员工的受尊重需要，应该让员工们感觉到他们在开发组织中很受尊重。对员工做出的成绩给予认同就是一种简便有效的方式。显然，也必须让员工们感觉到为他们所支付的报酬能够反映出他们的能力和经验的价值。

3. 最后，为了满足员工的自我实现需要，应该让员工对自己的工作负起责任，分配给他们一些要求较高（但不是不能完成的）的任务，并给他们提供培训和发展的机会以提高他们的技能。因为员工喜欢学习新的知识和技能，所以培训是一项重要的激励方式。

马斯洛的激励模型在一定程度上来说对人员管理是有帮助的，但是这个模型存在的一个问题是只从个人的角度考虑动力。模型没有充分考虑到一个人感觉他们是一个组织、一个专业性的团队、一个或多个文化组成中的一部分这个因素。身为一个有凝聚力的团队中的一员会使大多数人有强劲的工作动力。有充实的任务的人常常愿意去工作，因为他们被共同工作的人和从事的工作激发了积极性。因此，身为一个管理者，你要思考怎么将团队作为一个整体去激发他们的积极性。22.3 节将会探讨这个问题以及其他的一些团队合作的问题。

图 22-8 说明了一个管理者通常不得不面对的关于工作动力的问题。在这个例子中，一个非常有能力的小组成员对工作和小组都失去了兴趣。于是，其工作质量开始下降，变得不可接受。这种情况必须尽快得到处理。如果不能解决这一问题，小组的其他成员将会开始不满，觉得受到不公平的待遇。

案例分析：激励

Alice 是一个软件项目经理，她所在的公司主要研发报警系统。公司最近试图进入到快速发展的老年人和残疾人市场，开发帮助这个群体独立生活的技术产品。Alice 被指定领导一个 6 人研发团队开始这方面的项目，不再继续搞她所熟悉的报警技术。

Alice 的辅助技术项目进行得很顺利。在组内建立了良好的工作关系，一些具有创新的想法得到采纳。公司决定采用链接到网络的数字电视建立一个点对点的消息系统。然而，几个月之后，Alice 发现硬件设计专家 Dorothy 开始上班迟到，工作效率下降，而且没有和其他成员进行交流。

Alice 和小组的其他成员进行了非正式的谈话，试图了解 Dorothy 的个人环境是否发生了变化，是否影响到了她的工作。但是他们都不太清楚，所以 Alice 决定直接和 Dorothy 谈谈，了解其问题。

在几次否认之后，Dorothy 承认她对工作失去了兴趣。她希望能发展和运用她的硬件接口经验。但由于产品方向已经确定，在这方面她只有很少的机会。基本上她只是同其他团队成员一起作为 C 语言程序员在工作。

在她承认这项工作是一种挑战时，也同时明白在此她不能拓展擅长的接口技术。她很担心当进行完这个项目以后，将很难再去找到一个包含硬件接口的工作。因为她不想让团队知道她正在考虑下一份工作而影响到别人的工作热情，所以决定最好与团队中尽可能少的人谈论这件事。

图 22-8 个人激励

在这个例子中，Alice 试着去弄清楚 Dorothy 的个人情况是否有问题。私人问题通常会

影响到工作热情,因为人们不能专心于他们的工作。必须给他们时间并支持他们去解决自己的问题,同时也必须让他们清楚地认识到需要对雇主负责。

实际上 Dorothy 所面临的激励问题在项目开发与人们所预期的方向不一致时经常会发生。人们期望去做一种类型的工作,但实际上可能最终却干着完全不同的事情。在这个例子中,可能要决定让这个团队成员离开到别的地方寻找机会。而这里 Alice 试着去说服 Dorothy 拓展经历是职业生涯中一个正确的选择。她给了 Dorothy 更多的设计自主权,并组织了一些软件工程方面的培训课程,使 Dorothy 在结束当前的项目后能有更多的机会。



人员能力成熟度模型

人员能力成熟度模型 (People Capability Maturity Model, P-CMM) 是一个用于评估组织在管理员工自身发展方面的水平的框架。它强调在人的管理方面的最佳实践,为组织提供了改善它们人员管理过程的基础。

<http://www.SoftwareEngineering-9.com/Web/Management/P-CMM.html>

心理人格类型也会对激励产生影响。Bass 和 Dunteman (Bass and Dunteman 1963) 将职业人士分为 3 种类型。

1. 面向任务型,这类专业人员的动力来自于他们所从事的工作。在软件工程中,软件开发智力上的挑战激发了这类人员的工作热情。

2. 面向自我型,这类人的动力主要来自于个人成功和得到认可。他们更乐于把软件开发视为达到自己目标的手段。他们通常有更长远的目标,比如得到升迁,他们希望在工作中获得成功以帮助实现自己的这些目标。

3. 面向交互型,这类人的动力来自于同事们的存在和行为。随着用户界面的设计越来越受重视,面向交互的人们也越来越积极地参与到软件工程中来。

研究表明面向交互型人员通常喜欢小组作业,而面向任务型和面向自我型人员则通常喜欢独自工作。女人比男人更易成为面向交互型人员,她们通常是更有效的交流者。在图 22-10 中讨论了团队中这些不同的特性。

每个个体的工作动力由各种动力因素组成,但是在任一时刻总是只有一种动力居于支配地位。然而每个人都可能发生改变。举个例子,如果技术人员对所支付的报酬不满意,那么他很可能会变成面向自我型,把个人利益置于技术乐趣之上。如果团队协调得特别好,面向自我型的人也可能变成面向交互型的人。

22.3 团队协作

多数专业软件都是由一个项目团队开发完成的,这些团队的规模从两人到几百人不等。然而,让大型团队中的每个人都能有效地参与、解决一个问题,很显然是不可能的,通常要把这些大型团队分成小的项目小组,每个项目小组各负责整个系统的一个子项目。一个软件工程小组人员最好为 4~6 人,一定不能超过 12 人。分成小型的项目小组可以减少沟通中的问题。人人都互相认识,整个小组可以坐在一起开会讨论项目和当前正在开发的软件。

组成一个高效的项目小组是一项至关重要的管理任务。这个小组应该在技术、经验和

个性方面整体均衡。成功的小组并不只是各种技能均衡的个体的简单组合。好的小组具有一种凝聚力和团队精神，使得所有的成员既为自己的个人目标奋斗同时又为小组的成功而奋斗。

在有凝聚力的小组中，成员认为集体比个人重要。领导得当，凝聚力强的小组的成员都忠于小组，他们认同集体目标和其他成员。他们试图保护集体为一个整体，免受外来的干扰。这就使得小组足够健壮，有能力解决各种问题和突发状况。

建立有凝聚力的小组带来的好处包括：

1. 能够建立起小组自己的质量标准。因为这个标准的建立是经过小组一致同意的，与外部强加给小组的标准相比，更容易被大家遵守。
2. 成员互相学习、互相帮助。小组中的成员之间互相学习。鼓励互相学习可以消除由于相互不了解而引起的隔阂。
3. 知识能够共享。一旦有成员离开小组也能够保持工作的连续性。小组的其他成员可以接手关键的任务，确保不会对项目影响过大。
4. 鼓励重构和持续改进。不管一个设计或程序最初是谁创建的，小组成员都会团结工作以交付高质量的结果并解决问题。

优秀的项目管理者应该增强小组凝聚力。他们可以为小组成员及其家庭组织一些社会活动，可以通过给小组命名确立小组的特性和定位，尝试着建立小组的认同感，还可以开展有鲜明小组特色的小组建设活动，如运动和游戏。

增强小组凝聚力的最有效的方式是把组员当作自己人。必须认为小组成员是负责任的、可信赖的，保障小组成员的知情权。管理者们常常觉得某些信息不能让小组所有的成员都知晓，这样难免会产生相互之间的不信任。坦诚的信息交流是一种简便而又有效的方式，可以使小组成员感到他们的价值，感觉他们是小组的一部分。

在图 22-9 中的案例分析是一个例子。Alice 安排非正式的例会通知其他的小组成员将要做什么。她重视让成员们提出来自自己小组经验的新想法，从而参与到产品开发中来。“放松日”也是提高凝聚力的一个好方法。当人们互相帮助学习新技能时他们也一起得到了放松。

案例分析：团队精神

Alice 作为一个有经验的项目经理，了解建立一个有凝聚力的小组的重要性。当开发一个新产品时，她让产品描述小组和产品设计小组的所有成员都参与到探讨活动中来，与各小组中资深者一起对可能用到的技术进行讨论。她利用这样的机会使得项目团队中的每个成员都能够关心产品的描述和设计。她还鼓励老成员介绍新成员与其他小组中的成员认识。

Alice 每月都安排午餐会，这是项目组所有成员相互认识并谈论彼此所关心的问题的机会。在餐会上，Alice 告诉小组成员她所知道的公司的新闻、制度、策略等。然后每个成员简要总结他们已经完成和正在进行的工作，然后是小组讨论一般性话题，例如从年长同事那里获得的新产品构想。

每隔几个月，Alice 组织一次小组的“放松日”，用两天时间对小组进行“技术提升”。每个小组成员准备一项相关技术的最新内容并介绍给小组中其他成员。到一个较远的地方选择一个高级宾馆度假，给出充足的时间进行充分的技术讨论和成员彼此间相互沟通。

图 22-9 小组凝聚力

在一定程度上，一个小组能否有效率地工作取决于项目本身以及承担该项目的组织。如果这个组织不稳定，不断重组，工作无保障，处于一片混乱之中，团队成员很难专注于项目

开发。同样，如果一个项目不断改变或者面临取消的危险，成员们会对它失去兴趣。

给定一个稳定的组织和项目环境，以下3个要素对团队合作有着最重要的影响。

1. 小组中的人。项目小组需要不同类型的人员。因为软件开发包括各种活动：和客户谈判，编码，测试，编写文档等。

2. 小组的组织。应该这样组织团队：小组成员都能尽其所能，所承担的各项任务都能按时完成。

3. 技术上和管理上的交流。小组成员之间、软件开发团队和其他利益相关者之间的良好沟通是必不可少的。

正如所有的管理问题一样，获得合适的团队并不能保证项目成功。还会有太多的地方可能出现問題，如业务变更和业务环境改变。然而，不注意小组构成、组织以及交流的话，项目出现问题的可能性就大大增加了。

22.3.1 成员的挑选

管理者或团队领导者的任务是创建一个有凝聚力的小组并将其很好地组织起来，使其能一起有效地工作。这包括两方面：创建一个在技术技能和人格个性之间平衡的小组；组织成员一起有效地工作。有时，员工是从组织外面雇佣的；然而更常见的是，软件工程小组成员是从其他项目中有经验的现有职员中挑选来的。然而，事实情况是管理者很少有机会挑选小组成员。他们经常不得不使用公司里能找到的人员，即使这些人并不是这项工作的理想人员。

许多软件工程人员的动力主要来自于他们的工作。因而软件开发小组常常都是由一些对技术问题解决有独到见解的人员组成的。他们想尽可能地把工作做到最好，因此他们可能会刻意地重新设计系统或添加不在需求列表中的一些额外的系统功能，来达到他们所认为的改进目的。敏捷方法鼓励工程师在提升软件制品中起到主导作用。然而，有时候这就意味着时间被花费在一些不必要的事情上，并且有时工程师互相竞争着重写对方的代码。

技术性的知识和能力不应该作为挑选组员的唯一考虑因素。如果组员们有互补的工作动机，那么“工程师竞争”问题将得到缓解。以工作为动力的人很可能在技术上是最出色的。面向自我的人则可能善于推动整个工作的完成。面向交互的人有利于小组内部的交流沟通。所以，一个小组拥有面向交互型的人员特别重要。这种类型的人喜欢和别人交谈，可以较早地发觉小组中的紧张与不和谐状态，使得紧张与不和谐状态不致对小组造成严重影响。

图22-10中所示的案例分析中，表现了项目经理 Alice 如何去组建一个个性和谐的小组。这个特别的小组中有着面向交互和面向任务型两种人的良好组合；但是在图22-8中讲到了 Dorothy 面向自我的个性可能会导致一些问题，因为她干的并不是她希望的工作。Fred 作为领域专家的兼职角色也可能是个问题。他太过专注于技术的挑战，因而可能不能与其他小组成员进行良好的互动。他不能始终作为小组的一部分，这意味着他可能不能很好地与团队的目标相一致。

有的时候不可能挑选个性互补的成员组成一个小组。在这种情况下，项目管理者必须控制小组，不能让小组成员把个人目标凌驾于开发组织和小组的目标之上。如果所有的成员都参与到项目的各个阶段，这种控制比较容易实现。当项目小组成员在接受指示的时候不知道他们的任务在整个项目中的地位，他们很有可能会按照自己的意志行事。

案例分析：小组构成

在创建辅助技术开发小组的过程中，Alice 认识到选择个性和谐成员的重要性。面试的时候，她试着判定他们是面向任务型、面向自我型还是面向交互型的。她觉得自己主要是面向自我型的，因为她觉得这个项目是使得她能获得上层注意和得到提升的一个途径。因此她寻找一个或两个面向交互型的人，并期望能找到一些面向任务型的人来建立小组。她最后的看法是：

Alice—面向自我型
Brian—面向任务型
Chun—面向交互型
Dorothy—面向自我型
Ed—面向交互型
Fiona—面向任务型
Fred—面向任务型
Hassan—面向交互型

图 22-10 小组构成

举个例子，一个工程人员要为一个程序做编码设计，并注意到了一些可能的设计改善。如果在实现这些改善时，工程人员没有理解设计的初衷，那么任何改变，虽然初衷是好的，都会对系统的其他部分产生不利影响。如果整个小组从一开始就参与这项设计，他们就会理解为什么会做出这样的设计决定。他们就会认同这些决定，而不再反对它们。

22.3.2 小组的结构

小组的构成方式影响着小组的决策、信息交换的方式以及开发小组和小组外的项目利益相关者间的交流。项目管理者关心的重要组织结构问题如下。

1. 项目管理者应该是小组的技术负责人吗？技术负责人或者系统架构师负责项目开发过程中的重大技术决策。有些情况下，项目管理者有足够的技术和经验担当这个角色。然而，对于大型项目，最好将技术和管理角色分开。项目管理者指派一位资深的工程师担当项目架构师来负责技术领导任务。

2. 谁将参与做出重大技术决策，以及如何做出？应该让系统架构师、项目管理者还是让更多的小组成员一起做出？

3. 如何与小组外的利益相关者以及高层管理者进行交流沟通？在很多情况下，项目管理者将会在系统架构师（如果有的话）的辅助下负责这些沟通。另一个可以选择的组织模式是指派一个有着良好沟通能力的人担当此角色，专门负责外部联络。

4. 如何将分散的人员整合到一组？很普遍的情况是小组包括不同组织的成员，有的成员在共同的办公室办公也有人在家办公。在小组决策过程中应该将此考虑在内。

**雇用正确的人**

项目管理者通常负责挑选组织中的员工到他们的软件工程团队中。在此过程中获得可能的最好的员工是非常重要的，因为坏的选择决定会是项目的一个严重风险。

影响员工挑选的一些关键因素包括：教育背景和培训，应用领域和技术经验，沟通能力，适应性以及问题解决能力。

<http://software-engineering-book.com/web/people-selection/>

5. 小组怎么共享知识? 小组的组织方式影响信息的共享, 比起其他方式, 某些组织方式更有利于共享知识。然而, 也应该避免共享过多的信息, 以免超过成员承受的范围, 过多的信息会使成员在工作中分心。

小型的程序设计小组的组织结构通常是非正式的。小组领导和其他小组成员一起参与软件开发。在非正式结构的小组中, 完成哪些工作要经过小组集体讨论, 任务的分派按照个人能力和经验进行。高级的系统设计由小组的资深成员完成, 低级的设计则由承担具体任务的小组成员负责。

敏捷开发小组总是非正式小组。敏捷开发的拥护者认为正式的组织结构阻碍了信息交流。许多通常被认为应由管理层做出的决策, 如对项目进度的决策, 都交给了小组成员。然而, 依旧需要一个项目经理来负责战略决策和组外交流。

如果小组的大多数成员都既有经验又有能力, 非正式结构的小组会做得非常成功。小组的运作实行民主、集体决策。这样增强了小组的凝聚力和实力。然而, 如果小组的大部分成员经验不足, 或者能力不足, 非正式结构会阻碍小组的发展。没有一定的权威来指导工作的进行, 会使得小组成员间缺乏协调, 并有可能最终导致项目失败。

层次化小组中, 小组负责人处于最上层。负责人比起其他组员有更大的权限, 所以可以指导工作。组织层次结构清晰, 在此结构中, 上层做出决策, 而下层执行决策。在这种结构中, 交流主要是高层人员下达的指示, 从下层到上层的交流相对来说就很少了。

当一个被充分理解的问题能很轻易地被分解成可以由层次结构中的不同部分来开发的软件构件时, 层次化小组很有效。这种组织方式使得决策可以很快做出, 这是为什么军事组织遵循这一模型的原因。然而, 对于复杂的软件工程而言这一模型很少能奏效。在软件开发中, 在所有层次上有效的团队沟通是很重要的。

1. 软件的变更通常会引起系统多个部分的变化, 因此需要层次结构中各个等级的人员一起讨论磋商。

2. 软件技术飞速发展, 年轻职员往往比有经验的老职员对技术懂得更多。自上而下的交流方式意味着项目管理者找不到机会使用新技术。而且由于在开发中使用在年轻员工看来是过时的技术, 越来越多的新工会产生挫败感。

项目管理者面临的主要挑战就是小组成员技术能力的差异。最好的程序员可能比最差的程序员的工作效率高 25 倍。因此, 要使这些“超级程序员”发挥出最大的作用, 就需为他们提供尽可能多的支持。与此同时, 专注于这些“超级程序员”会使其他组员失去动力, 他们会因为没被分配到任务而不满。因为他们担心这可能关系到他们的职业生涯发展。此外, 如果一个“超级程序员”离开了公司, 会对项目产生巨大的影响。因此, 采取一个基于个别专家的小组模型会造成重大的风险。



物理工作环境

人们所处的工作环境既影响小组沟通也影响每个人的工作效率。独立工作间对从事复杂的技术工作的人集中注意力特别有帮助, 这样技术人员就会最大限度地受到干扰。然而, 共享空间特别有助于沟通。设计良好的环境应该综合考虑到这两方面的需要。

<http://software-engineering-book.com/web/workspace/>

22.3.3 小组的沟通

组员相互之间以及和其他项目利益相关者之间的准确高效的沟通是绝对必要的。组员之间需要交流各自工作的情况、做出的设计决策,以及对原先设计决策进行修改。组员必须解决其他利益相关者所提出来的问题,并告之在系统方面、小组方面以及在交付计划方面等的变更。良好的交流有助于增强小组的凝聚力。组员开始理解小组中其他人员的动机、长处以及弱点。

影响沟通有效性和效率的主要因素有:

1. 小组规模。随着小组规模的扩大,要保证所有的成员都能彼此有效交流就变得很困难了。单向交流链的数目是 $n \times (n - 1)$, 其中 n 代表小组的人数,可以看出,对于一个有 8 名成员的小组,有 56 条可能的交流路径。有些成员之间很可能没有实际交流。小组成员之间职位上的差异意味着沟通常常是单向的。管理者和经验丰富的工程师在与经验较少的成员进行交流时,容易处于支配地位,谈话的对象常常不愿主动开始交谈。

2. 小组结构。在一个没有正式组织结构的小组中,员工的沟通比有正规层次化结构的小组中员工的沟通更为有效。在有层级结构的小组中,沟通是按照层级结构进行的。处于同一层级的人可能彼此不进行交流。这是含有几个开发小组的大型项目中的一个突出问题。如果开发不同子系统的人员只与他们的管理者进行沟通,通常会造成项目的延期和理解上的错误。

3. 小组构成。如果小组中人太多并且个性类型相同(在 22.2 节讨论过),这样可能容易发生冲突,沟通也不能正常进行。由混合性别组成的小组(Marshall and Heslin 1975)中的沟通要比由单一性别组成的小组容易进行。女性比男性更容易成为面向交互型的人,小组中的女性成员可作为小组沟通的控制员和协调员。

4. 物理工作环境。工作场所的结构是推动或阻碍沟通的主要因素。一些公司为员工提供标准的开放式办公室;其他一些公司使用了包含了私人工作区域与小组工作区域的混合式工作空间,这使得兼顾协作的活动和需要高度集中的个人开发成为可能。

5. 可用的沟通渠道。交流的形式有很多——面对面的方式,发电子邮件,下达正式文件,电话以及技术(比如社交网络和 Wiki)。项目组分布在各处的情况越来越常见,组员都在远地工作,需要广泛采用能使小组交流变得便捷的交互技术,如会议系统。

对项目管理者来说,最后期限通常都很紧,所以他们更愿意使用不太费时的交流渠道。他们依靠会议以及正式文件向项目职员和利益相关者传达信息,并向项目成员发送内容很长的邮件。虽然从一个项目管理者的角度来看这是有效率的交流方式。但通常却没有好的效果。我们可以发现有很多原因使得人们不去参加会议,所以他们听不到报告。人们没有时间去阅读过长而且与自己的工作没有直接关系的文档。当同样的文件产生了几个版本的时候,读者会很难找到其中的不同点。

当沟通是双向的时候,有效的沟通就达到了。有关的人员讨论问题和咨询,并且建立起对提议和问题的共同的理解。可以通过会议达到双向沟通目的,虽然会议通常被强势的人员所控制。管理人员与小组成员在咖啡时间进行的非正式讨论有时更加有效。

越来越多的项目团队中包括在外地工作的人员,这使得举行会议更加困难。为了解决交流问题,可以使用 Wiki 和博客来支持信息交流。Wiki 支持合作创造和编辑文档,博客支持主题为线索的讨论,组员可以提出问题和评论。无论在哪,Wiki 和博客允许项目成员和外

部利益相关者交换信息。它们能帮助管理信息，分辨讨论的类型，而这些使用电子邮件的时候经常把人搞糊涂。当然，也可以使用即时通信工具和电话会议处理需要讨论的问题。这是很容易安排的。

要点

- 软件工程项目要想按进度、按预算进行开发，完善的软件项目管理是必要的。
- 软件项目管理与其他的工程管理有明显区别；软件产品是无形的；软件项目可能很独特或者有创新，这样就没有实在的经验以供借鉴；软件过程没有传统工程过程那么成熟。
- 风险管理包括识别并评估重大的项目风险，从而判断这些风险发生的可能性及其后果的严重程度。对很有可能发生并有潜在严重性的风险，应该制订有关规避、管理和解决可能发生的风险的计划，包括对风险发生的分析和当风险发生时的应对措施。
- 人员管理涉及为项目选择正确的人员、组织团队和工作环境，这样他们就能尽可能高效地工作。
- 对一个人的激励因素会包括与其他人的交互作用、得到管理层和同行的赏识，以及得到个人发展的机会等。
- 软件开发小组不宜太大且要具有凝聚力。影响小组效率的关键因素是小组中的人员、小组组织的方式以及组员之间的沟通。
- 一个小组内部的沟通受许多因素的影响，如小组成员的职位、小组的规模、小组的性别组成、小组成员的个性以及可用的沟通渠道。

阅读推荐

《The Mythical Man Month (Anniversary Edition)》自 20 世纪 60 年代以来，软件管理的难题一直就没有变化，这是软件管理方面的最好的书之一。该书有趣、易懂，描述了最早的大型软件项目之一的 IBM OS/360 操作系统的管理活动。这个纪念版（在 1975 年发表的最初版本之后 20 年出版）还收录了 Brooks 的其他一些经典的论文。（F. P. Brooks, 1995, Addison-Wesley）

《Peopleware: Productive Projects and Teams, 2nd ed》是一本经典著作的新版本，强调了软件项目管理中人文关怀的重要性。这本书容易阅读，是少有的认可人的工作场所的重要性的书之一。极力推荐此书。（T. DeMarco and T. Lister, 1999, Dorset House）

《Waltzing with Bears: Managing Risk on Software Projects》是一本实践性很强且易读的书，主要介绍风险和风险管理。（T. DeMarco and T. Lister, 2003, Dorset House）

《Effective Project Management: Traditional, Agile, Extreme. 2014 (7th ed.)》是一本关于通用项目管理的书，并不局限于软件项目管理。它基于被称作 PMBOK（Project Management Body of Knowledge）的项目管理知识体系。并且与含有这个内容的许多书不同，它探讨了敏捷项目中项目管理的技术。（R. K. Wysocki, 2014）

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap22/>

支持视频的链接: <http://software-engineering-book.com/videos/software-management/>

练习

- 22.1 解释软件系统的无形性为什么给软件项目管理提出了特殊的难题。
- 22.2 解释最好的程序设计者为什么不一定能成为最好的软件管理者。22.1 节中列出的管理活动可以帮助你回答这个问题。
- 22.3 根据文献中已报告的项目问题的实例，列出这些失败的程序设计项目中存在的管理难点和错误（可参考“阅读推荐”中 Brooks 的书）。
- 22.4 除了图 22-1 中所列的风险，识别可能在软件项目中出现的其他 6 种风险。
- 22.5 为什么在一个项目开发的过程中风险发生的可能性不断上升，而且这些风险造成的后果很可能会变化？
- 22.6 价格锁定的合同，即承包人以某个确定价格投标一系统开发，是一种将项目风险从委托人转移给承包人的做法。如果出现问题，需要由承包人承担。分析一下为什么这种合同容易增加产品风险的可能性。
- 22.7 为什么让小组所有成员都知悉项目的进展情况和技术上的决策能够增强小组的凝聚力？
- 22.8 在极限编程团队中许多管理决策权被下放到团队成员手中，这会带来哪些问题？
- 22.9 编写一个类似本书风格的案例研究，证明项目组中沟通的重要性。假定这个项目组一些组员在外地工作，不可能在短时间内召集在一起。
- 22.10 你的经理要求你在某一时刻之前交付软件产品，而据你了解只有让你的项目组无偿地加班加点，才能赶上进度，而项目组所有的成员都有孩子需要照顾。你应该接受经理的要求，还是应该说服你的项目组成员加班？做出你的决定要考虑哪些重要因素？

参考文献

- Bass, B. M., and G. Duntzman. 1963. "Behaviour in Groups as a Function of Self, Interaction and Task Orientation." *J. Abnorm. Soc. Psychology* 66 (4): 19–28. doi:10.1037/h0042764.
- Boehm, B. W. 1988. "A Spiral Model of Software Development and Enhancement." *IEEE Computer* 21 (5): 61–72. doi:10.1109/2.59.
- Hall, E. 1998. *Managing Risk: Methods for Software Systems Development*. Reading, MA: Addison-Wesley.
- Marshall, J. E., and R. Heslin. 1975. "Boys and Girls Together. Sexual Composition and the Effect of Density on Group Size and Cohesiveness." *J. of Personality and Social Psychology* 35 (5): 952–961. doi:10.1037/h0076838.
- Maslow, A. A. 1954. *Motivation and Personality*. New York: Harper & Row.
- Ould, M. 1999. *Managing Software Quality and Business Risk*. Chichester, UK: John Wiley & Sons.

项目计划

目标

本章的目的是介绍项目计划、进度安排以及成本估算。阅读完本章后，你将：

- 理解软件成本计算的基础以及影响为外部客户所开发的软件系统定价的因素；
- 了解在计划驱动的开发过程中的项目计划应该包含哪些部分；
- 理解项目进度安排包含的内容以及如何使用条状图制作项目进度安排；
- 了解基于“计划游戏”的敏捷项目计划；
- 理解成本估算技术以及如何使用 COCOMO II 模型进行软件成本估算。

项目计划是软件项目管理者最重要的工作之一。项目管理者必须将工作分解开来并分配给团队成员，必须预见可能出现哪些问题，并且准备好相应的试探性的解决办法去应对这些问题。项目计划是在项目的开始建立的，并且随着项目的进度更新，是用于说明工作如何开展，以及估计项目进度的。

项目计划发生在项目生存周期的以下 3 个阶段。

1. 投标建议书阶段，即为争取一个软件系统开发合同的投标阶段。在这个阶段，需要做出计划以帮助管理者判断是否有完成这项工作所需的资源，并计算出向客户开出的价格。

2. 项目开始阶段，此时需要做出如下计划：确定参加此项工作的人员；将工作分解成若干个子项目；在公司中如何分配资源等。此时，已经掌握了比投标阶段更多的信息，因此可以进一步完善最初的工作量估计。

3. 贯穿于项目过程中，定期地更新计划来反映关于软件及其开发的新信息，这样项目管理者对开发的系统和开发团队的能力就了解越多。随着软件需求的改变，工作分解需要修改，项目计划需要扩展，这些信息能帮助项目管理者越准确地估计项目的开发时间。

通常，由于在投标阶段没有要开发的项目的完整需求，不可避免，建议书阶段的计划只是预测性的。但是必须按照客户要求拿出建议书，对基于所要求的软件功能做出高水平描述。计划通常是建议书要求的一部分，所以，必须做出对于开展工作可信的计划。如果能够赢得合同，通常必须重新计划项目，将建议书完成之后所出现的变更以及关于软件、开发过程、开发团队的信息考虑在内。



日常成本

当你估计在软件项目上的工作量成本时，不能简单地用人员单位时间的工资乘以投入到项目的时间。你还必须考虑所有的组织日常开支（办公空间，管理等），这些都应该都是项目收入所承担的。通过计算这些日常开支，加上项目中工程师的成本乘上一个系数，就是项目的总的成本。

<http://software-engineering-book.com/web/overhead-costs/>

在投标争取一个合同的时候,必须计算出需向客户提出的开发软件的价格。计算价格首先需要估计完成项目所需要的成本。这包括计算完成各个活动所需要的工作量,然后计算所有活动的工作量。项目管理者必须客观地计算软件成本,才能准确地预测开发软件的成本。一旦对可能的工作量有合理的估计,接下来才能计算出价格,向客户提出报价。但是正如下一节要讲的,这不只是简单地成本+利润,还有很多因素影响软件项目的定价。

在计算软件开发项目总成本时要使用以下3个主要参数:

- 工作量成本(支付给软件工程师和管理人员的费用);
- 包括硬件维护和软件支持在内的硬件和软件费用;
- 差旅费和培训费用。

对于绝大多数项目,主要的成本是工作量成本。项目管理者必须估算完成此项工作可能需要的总成本(人月)。很显然,所掌握的用于估算的信息是有限的,所以必须首先做出最贴近的估算(往往是偏于乐观的),然后加上一个很大的应急开支(额外的时间和成本)。

对于商业系统来说,正常的情况下会使用相对便宜的商品硬件。然而,假如需要得到中间件和平台软件的授权的话,软件成本就会大大增加。如果项目是在不同地方开发,可能还需要长途出差。虽然差旅费通常只是工作量的一小部分,但是花在旅途上的时间就被浪费掉了,极大地增加了项目的工作量成本。可以使用电子会议系统和其他协作软件减少差旅,从而有更多的时间进行有成效的工作。

一旦获得开发系统的合同后,应该立刻精练该项目的框架计划,创建项目的启动计划。这个阶段管理人员需要知道更多的系统需求。此时的目标是创建一个能够帮助项目人员决策补充和做出预算的、包含足够细节的项目计划。将这个计划作为基础,在组织内部分配项目资源,并且帮助决定是否雇用新员工。

上述计划也应包括项目监控机制。项目管理者必须时刻追踪项目的进展,比较实际的进展和成本与原计划的进展和成本的不同。大部分的组织都有正式的监控机制,但是从监控来说,一个好的项目管理者应该能够从与员工的非正式交谈中对项目的情况建立起清晰的认识。非正式监控能够在困难出现时及时发现,从而预测到潜在的项目问题。例如,在与员工的日常交谈中可能暴露出团队在交流系统中存在软件故障。从而项目管理者可以立刻安排一个交流专家帮助发现和解决该问题。

项目计划在开发过程中总是随着需求变更、技术问题和开发中所出的问题而改变。要保证项目计划是一份有用的文件,帮助员工理解要达到的目标和软件交付的时间。因此,随着软件开发的推进,需要修改进度安排计划、成本估计以及风险估计。

如果使用敏捷方法,仍然需要项目启动计划。不论使用的是什么方法,公司仍然需要计划如何给项目分配资源。然而,这并不是一个详尽的计划,只需要包括关于工作分解和项目日程表这些必要的信息。在开发过程中,要为每个软件版本制订一个非正式的项目计划和工作量成本估计,应该让团队所有员工都参与到计划过程中来。敏捷开发的一些方面已经在第3章中涉及,其他部分将在23.4节中进行讨论。

23.1 软件报价

原则上,对于客户来说,软件产品价格只是简单的开发成本加利润。不过,实际上项目成本和向客户提出的价格之间的关系并不总是这样简单。软件报价必须考虑到方方面面的因素,如组织因素、经济和政治因素、商业上的因素等。必须考虑的因素如图23-1所示。必须考虑组

织上的问题、项目相关的风险，以及将要使用的合同的类型。这些都将引起价格上升或者下降。

因 素	描 述
合同条款	客户可能会愿意让开发者保留对源代码的所有权，并在其他项目中复用。这样所收取的价格可以降低以反映源代码对于开发人员的价值
成本估算的不确定性	如果组织对成本估算不太确定，它可能增加应急开支项，使得提出的报价超出了一般收益
财务状况	处于财务困境中的公司可能会降低报价来得到一份合同。比正常情况下少赚些甚至亏一点也比没有项目好。在困难时期现金流比利润更重要
市场机遇	开发组织可能为进入一个新的软件市场而提出一个低的报价。在一个项目上的低回报可能会换来今后更大收益的机会，而且获得的经验有助于开发新产品
需求易变性	如果需求可能会发生改变，组织会降低它的价格以得到合同，在合同签订后，需求的改变将带来高的报价

图 23-1 影响软件报价的因素

为了说明项目定价的一些问题，考虑下面的场景。

PharmaSoft 是一家小型软件公司，雇用了 10 个软件工程师。公司刚完成了一项大工程，但是现在得到的合同仅需要 5 个开发人员就够了。但公司正在竞标一个主要的制药公司的大合同，需要在 2 年内完成 30 人年的工作。项目在至少 12 个月内不会动工，但一旦获得批准，该项目会让公司的财务状况有很大起色。

PharmaSoft 公司得到一个机会竞标一个需要 6 个人且在 10 个月内完成的项目。成本（包括项目的日常开支）估计在 120 万美金。但为了提高竞争力，PharmaSoft 公司给顾客的报价是 80 万美金，这就意味着尽管在这个合同中损失了一些收入，但它为一些在一年时间内就会开始的更有利可图的项目保留了高水平的职员。

这是一种称为“赢得合同的报价”的软件报价方法的例子。“赢得合同的报价”意思是软件公司对于客户期望的报价有所了解，继而根据客户期望的价格投标。这看似不道德而且不严谨，然而对于客户和系统开发方双方都有一些好处。

项目成本是在项目建议书的基础上达成共识的。开发商和客户之间通过协商来建立详细的项目规格说明，该规格说明受到双方认可的成本的制约。买方和卖方必须对什么是可接受的系统功能取得共识。在许多项目中，不变的因素是成本而不是项目需求。为了让成本处于预算范围之内可能会改变需求。

举个例子，OilSoft 公司投标为某石油公司开发一个新的油料传输系统，该系统为石油公司的加油服务站安排油料进度。由于没有关于该系统详细的需求文件，开发者估计 90 万美元的价格可能具有竞争力，并且在石油公司的预算之内。在签订合同之后，OilSoft 公司对系统的详细需求进行协商以交付基本的功能，然后再估算其他需求带来的成本。

这种方法对于软件开发人员和客户都有好处。通过协商避免了实现困难和非常昂贵的需求。灵活的需求能够更容易地复用软件。这家石油公司将合同授予了一家可信任的公司。甚至，项目的成本有可能被分散到系统的多个版本之中。这样能减少系统部署的费用，并且允许客户跨数个财政年度进行项目成本预算。

23.2 计划驱动的开发

计划驱动的开发或者是基于计划的开发，它是一种给开发过程制订详细的计划的软件工程方法。首先是要创建项目计划。项目计划完整地记录以下内容：要完成的工作，谁将执行此项

工作，开发进度安排，以及项目的成果是什么。管理者使用计划支持项目决策并将其作为衡量项目进展的方法。计划驱动开发基于工程项目管理技术，可以看作管理大型软件开发项目的传统方法。敏捷开发包含一个不同的计划过程，其中决策将被推迟，具体将在 23.4 节中进行讨论。

计划驱动的开发的的问题是，由于软件开发和使用的环境的变化，必须修改许多早期的决策。推迟计划决策能够避免不必要的重复工作。赞同计划驱动的理由是，早期计划能够很好地考虑组织问题（组织员工等），能够在项目开始前发现潜在的问题和依赖性，而不用等到项目开发时才发现。

在笔者看来，项目计划最好的方法是将计划驱动方法和敏捷开发结合起来。其中的平衡取决于项目的类型和人员的技术水平。在极端的情况下，大型信息安全和安全关键性系统需要大量的前期分析，在投入使用前必须万无一失。这种系统大部分应该是计划驱动的。在另一种极端情况下，部署在快速变化环境中的小型或中型信息系统应该使用敏捷方法开发。当几个公司合作开发项目时，通常使用计划驱动的方法协调各个开发地点的工作。

23.2.1 项目计划

在计划驱动的项目开发中，项目计划包括项目可用资源的分配、工作分解以及完成工作的进度安排。计划应该指出开发的项目和软件的风险以及用于风险管理的方法。项目计划书的具体内容随着项目和开发组织类型的不同而改变。不过，多数的计划书应该包括以下几个部分。

- 1. 引言。这一部分简要论述项目的目标，并列出具影响项目管理的种种约束条件，如预算、时间的限制等。
- 2. 项目组织。这一部分阐述开发团队的组织方式、人员构成及其分工。
- 3. 风险分析。这一部分分析项目可能存在的风险以及这些风险发生的可能性，并提出降低风险的策略（在第 22 章中讨论）。
- 4. 硬件和软件资源需求。这一部分介绍完成开发所需的硬件和支持软件。如果需要购买硬件，应注明估算的价格和交付的时间。
- 5. 工作分解。这一部分把项目分解成一系列的活动，并且识别每个项目活动的输入和输出。
- 6. 项目进度安排。这一部分要描述项目中各活动之间的依赖关系。到达每个里程碑预期所需的时间以及人员在活动中的分配。本章下一节将讨论进度安排可能的表示形式。
- 7. 监控和报告机制。这一部分说明要提交哪些管理报告、什么时候提交，以及使用什么样的项目监控机制。

主项目计划应当总是包含项目风险评估和项目进度安排。此外，还要制订多个补充计划，用于支持其他过程活动，例如测试和配置管理。图 23-2 给出了一些可能使用的补充计划。在开发大型复杂的项目时需要使用这些补充计划。

计 划	描 述
配置管理计划	描述所要采用的配置管理规程和结构
部署计划	描述软件和相关的硬件（如果需要）在客户的环境下会怎样部署。其中应该包含一个从现有系统迁移数据的计划
维护计划	预测维护需求、成本以及工作量
质量计划	描述在项目中所要使用的质量过程 and 标准
确认计划	描述用于系统确认的方法、资源和进度安排

图 23-2 项目计划补充

23.2.2 计划过程

项目计划是一个迭代的过程，在项目的启动阶段初始项目计划的创建就开始了。图 23-3 是 UML 活动图，给出了典型的项目计划过程的工作流。计划不可避免地会改变。由于在项目进行期间不断产生新的关于系统和项目团队的信息，所以必须经常性地修正原有的计划，反映需求、进度安排以及风险的变更。业务目标发生改变，也会导致项目计划相应地改变。业务目标发生改变，会影响到所有的项目，这些项目就必须重新计划。

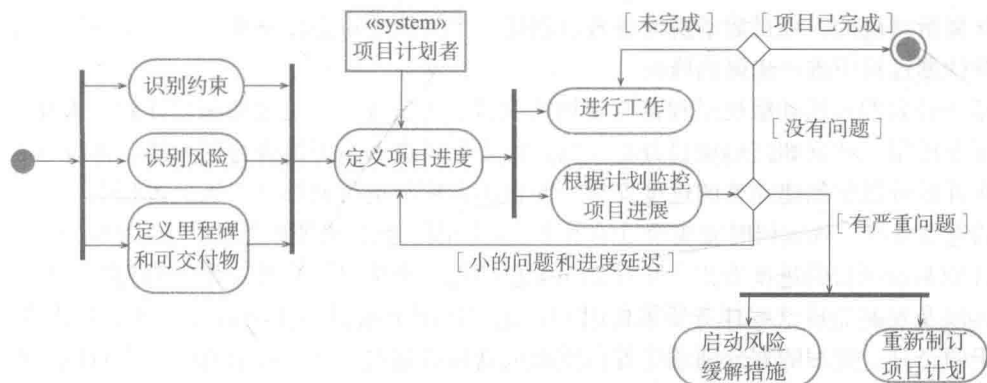


图 23-3 项目计划过程

在计划过程的开始阶段，应该从评估影响项目的各种约束条件开始。这些约束条件包括项目的交付日期、现有的人员情况、总体预算、可用工具等。另外还要定义项目的里程碑和可交付物。里程碑是进度安排中的能够估计项目进展的那些位置点，比如移交系统进行测试。可交付物是交付给客户的项目成果（比如系统的需求文档）。

接下去计划过程进入一个循环直到项目完成时结束。拟定项目的进度安排，并启动进度安排中的各种活动或继续某些活动。在一段时间之后（通常约 2～3 星期），检查项目的进展状况，注意项目进展与进度安排的偏差。因为最初对项目参数的估算肯定是近似的，存在小的拖延是正常的，所以计划需要不断地修改。

当定义一个项目计划时，应该做出现实的而不是乐观的假设。在一个项目中一些问题会不停地出现，这将导致项目的延期。项目开始时的假设和进度安排应该适当保持悲观，并且将非预期的问题也考虑进来。在拟定计划时应该把各种偶然因素考虑进去，这样假如发生问题，交付进度不会被严重地中断。

如果开发工作发生严重的问题，可能导致项目很大的延期，就需要采取风险缓解措施减少项目失效的风险。除了这些措施，还需要重新计划项目，可能就项目的种种约束条件和可交付物的有关事宜，重新与客户协商。应该重新建立新的何时完工的进度安排，并征得客户同意。

如果协商不成或者风险缓解措施没有效果，就应该安排正式的项目技术评审。评审的目标是找到能够使得项目继续进行的替代方法，评审也要检查客户的目标是否发生变化以及项目是否与目标一致。

评审（review）的结果可能是做出取消项目的决定。这可能是技术上或者管理上的失败。但是，更可能是外部变化影响项目的结果。大型软件的开发时间往往长达数年。在开发过程中，业务目标和企业优先考虑的事情不可避免地会发生变化。如果这些变化表明不再需要这套软件或者原始项目需求不妥，管理人员可能决定停止软件开发，或者对项目做出重大改变

以反映组织的目标变化。

23.3 项目进度安排

项目进度安排是决定如何组织项目工作，将其分割成单独的一个个任务，并且何时以何种方式完成各项任务的过程。需要估算需要用于完成各个任务的时间、需要的成本以及完成这些既定任务的人员。除此之外，还必须估算完成每项任务所需要的硬件和软件资源，比如在开发嵌入式系统时，要估算在专用硬件上需要的时间和运行一个系统模拟器的成本。正如本章开篇所讨论的，项目启动阶段通常会创建一个初始项目进度安排。此进度安排会在后续的开发计划过程中进一步得到修改。

基于计划的过程和敏捷过程都需要初始项目进度安排，只是敏捷项目计划不太详细。初始进度安排用于计划如何给项目分配人员，检查项目进展是否符合合同承诺。传统开发过程中，在开始阶段就创建完整的进度安排，并且随着项目进行而修改。敏捷过程中，必须有一个总的进度安排，确定何时完成项目的各个主要阶段。然后再使用迭代的方法计划各个阶段。

计划驱动项目的进度安排（见图 23-4）包括把一个项目所有的工作分解为若干个独立的任务，以及判断完成这些任务所需的时间。正常情况下项目的各项活动应该至少持续一周，并短于两个月。更细的划分则意味着在重新计划和更新项目计划时会花掉太多时间。对所有项目活动安排的最高的时间期限约为 6 ~ 8 周。如果一项活动持续的时间超出这个范围，就应该在项目计划和进度安排中再次细分。



图 23-4 项目进度安排过程

有些任务是并行进行的，不同的员工研发系统不同的部件。负责进度安排的人员必须协调这些并行任务并把整个工作组织起来，从而使工作量资源得到充分利用。同时尽量避免各个任务之间不必要的依赖性。一定要避免出现因一项关键任务没有完成而使整个项目延期交付的情形。

如果一个项目在技术上非常先进，即使管理者把所有可能的意外都考虑进去，初始的估算也肯定是偏乐观的。在这一点上，软件进度安排与任何其他类型的大型高技术项目没有什么不同。新的飞机、桥梁甚至新型的汽车因为意想不到的问题常常延期交付。因此，随着有关项目进展信息的增多，必须不断地更新项目进度安排。如果进行进度安排的项目与原来某项目相似，可以沿用原来的进度安排。事实上，由于不同项目可能使用不同的设计方法和使用不同的实现语言，先前项目的经验可能不适用于计划新项目。

在估算进度时，管理者应该考虑到项目出现问题的可能性。比如，做这个项目的个别人员可能生病或离职，硬件可能会崩溃，所需的基本的支持软件或硬件有可能迟迟不能交付。如果这是个新项目并且技术先进，其中某些部分可能比原来预期的要困难得多，花费的时间也多。

一个好的经验法则是，进行估算时先假定什么问题也没有，然后再把预计出现的问题加到估算中去。其他的偶然因素可能带来意想不到的问题，在估算时也可以考虑进去。这些偶

然因素是由项目的类型、过程参数（最后期限、标准等）以及该项目的软件工程人员的素质和经验决定的。意外情况可能使项目需要的成本和时间增加 30% ~ 50%。



活动图

活动图是项目进度安排的一种表示方法，并以有向图（directed graph）的形式展示了项目计划。它给出那些能并行执行的任务和那些必须按顺序执行的任务，这些都是根据某任务对先前的任务的依赖关系做出的。如果一个任务依赖于几个其他任务，那么所有被依赖关系的任务都必须在此任务开始之前完成。活动图中所谓的“关键路径”是最长的依赖任务序列。它决定了项目的持续时间。

<http://software-engineering-book.com/web/planning-activities/>

23.3.1 进度安排表示方法

项目进度安排可简单地用一个表来表示。列出任务、估计工作量、工期、任务依赖关系（见图 23-5）。但是这种表示方式很难发现不同活动之间的关系和依赖性。所以，人们研究出了另外一些更容易阅读和理解的图形可视化方法。通常使用的有以下两种表示方法：

- 1. 基于日历时间的条状图，可以表示每项活动的负责人是谁、预计所用的时间，以及该项活动预计的开始和结束时间。条状图有时也叫甘特图，是以发明人 Henry Gantt 命名的。
- 2. 活动网络，表示构成项目的不同活动之间的依赖关系。这些网络在相应的在线章节中介绍。

任务	工作量（人日）	持续时间（天）	依赖关系
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

图 23-5 任务、持续时间以及依赖关系

项目活动是基本的计划单元，每个活动有：

- 以天或者以月计算的工期（持续时间）；
- 工作量估计，反映完成工作所需的人日或人月数；
- 活动完成的最后期限；
- 定义好的终点，可以是一份文档、举行评审会或者是成功执行了所有测试等。

在进行项目计划时，应该建立项目里程碑。一个里程碑就是一个项目阶段的逻辑终点，在这个项目阶段中项目进度可以接受评审。每个里程碑应该用一个简短的报告（通常是一封邮件）记录下来，总结已经完成的工作以及工作是否按照计划完成。里程碑可能是关于单个的任务或者是一组相关联的活动。例如，在图 23-5 中，里程碑 M1 和任务 T1 有关，并且标记了该活动的结束。而里程碑 M3 和两个任务 T2、T4 有关；在任务结束的时候没有单独的里程碑。

一些活动会形成项目可交付物——作为产品交付给软件客户的输出。通常情况，可交付的文档是在项目合同中明确说明的，在客户看来项目的进展表现为这些可交付的文档。里程碑和可交付物并不相同。里程碑是用于汇报进度情况的简短报告，而可交付物是更具体的项目输出，比如需求文档或系统初始实现。

图 23-5 展示了一组假想的任务、估计的工作量投入和持续时间，以及任务之间的依赖关系。从图 23-5 可以看出任务 T3 依赖于任务 T1，也就是说 T1 必须要在 T3 开始前完成。例如，T1 必须因为系统复用而进行选择，而 T3 则对被选择的系统进行配置。在选择和安装将被修改的应用系统之前，不能开始系统的配置。

可以看出一些活动估计的持续时间比必需的工作量长；而有些则相反。如果工作量比持续时间少，这就表示分配到这个任务的人员没有全职工作。如果工作量超过持续时间，就表示几个成员同时进行这个任务。

图 23-6 是使用图 23-5 中的信息以条状图的形式表示项目的进度安排，表示了项目的日程安排和各项任务的开始和完成日期。从左往右阅读，条状图清晰地表示任务何时开始以及何时结束。条状图中也标明了里程碑（M1、M2 等）。可以看出独立的任务是并行执行的（比如任务 T1、T2 和 T4 都在项目的起始处开始执行）。

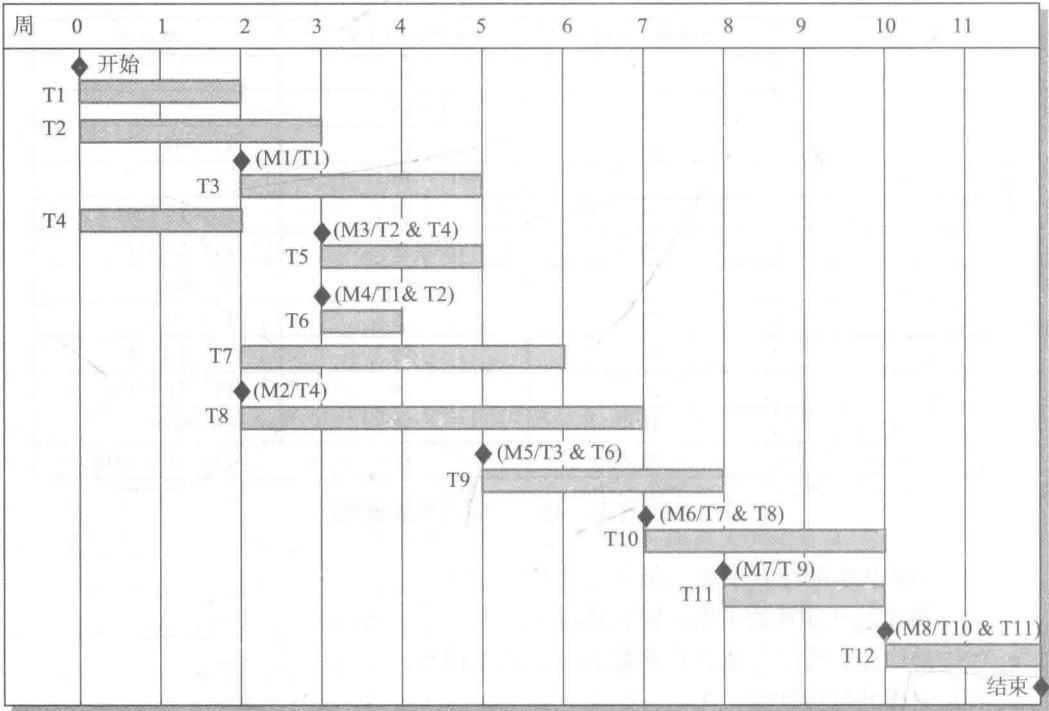


图 23-6 活动条状图

除了考虑进度安排外，项目管理者还要考虑资源的分配，尤其是参加项目活动的人员的分配，以及给他们分配项目任务。可以用项目管理工具对资源分配进行分析，生成条状图，从而显示在哪些时间段上雇用哪些职员（见图 23-7）。项目职员可能同时承担多个任务，或者有时他们并不在项目中工作。在这期间他们可以休假、做别的项目或参加培训。图 23-7 中的兼职任务使用对角线在任务条上标识。

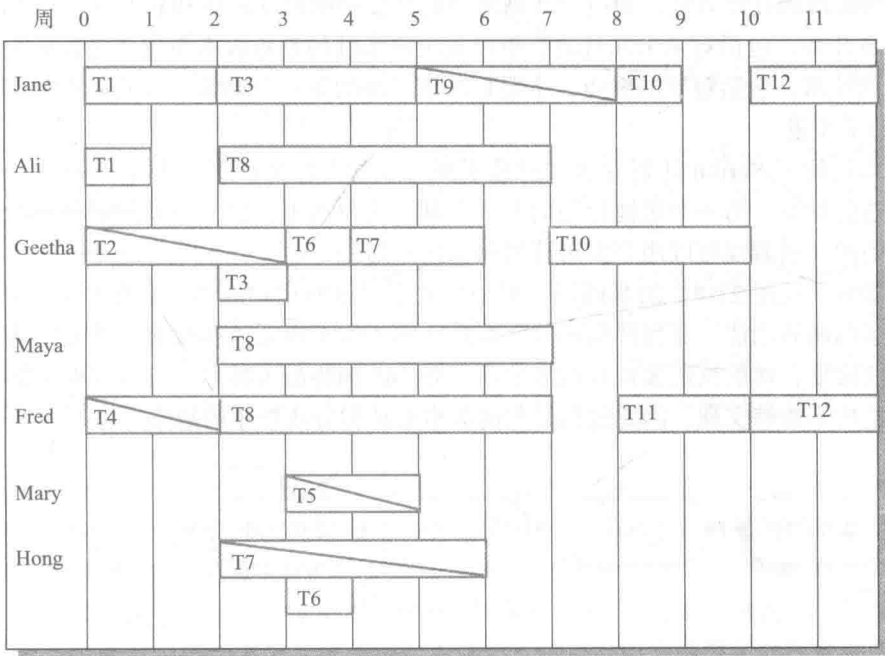


图 23-7 人员分配图

大型的开发组织通常会根据项目需求聘请许多专家，这有可能影响项目的进度安排。在图 23-7 中，我们可以看到，Mary 是一位专家，只参加项目中的一个任务（T5）。在开发复杂的系统时，专家的使用是不可避免的，但是这有可能引起进度安排问题。如果有专家参与的一个项目延迟了，就会给其他也需要专家的项目带来连锁反应。因为专家无法到位，这些项目很可能也要延迟完成。

如果任务延期了，将明显地影响以后的依赖于它的其他任务。只有延期的项目完成后，其他项目才能开始。任务延期会引起严重的人员分配问题，特别是在人员同时参与几个项目的情况下。如果任务 T 延期，可能将分配给任务 T 的人员分配其他工作（任务 W）。完成任务 W 可能比任务 T 的时间长，但是人员一旦分配出去，就不可能轻易地将他们重新分配回原来的任务（T）。由于等待任务 W 的完成，这将进一步导致任务 T 的延期。

通常情况下，应当使用一个项目计划工具（例如 Basecamp 或 Microsoft Project）来创建、更新和分析项目进度信息。项目管理工具通常需要将项目信息输入到一个表格里，而这些工具会创建一个项目信息的数据库。接着，这些工具可以自动从数据库中生成条状图和活动图。

23.4 敏捷计划

软件开发中的敏捷方法是一种迭代的方法，软件是以增量式方式进行开发和交付给客

户。和计划驱动方法不一样的是，这些增量中的功能不是提前计划好的，而是在开发过程中决定的。决定增量包中包含何种功能取决于项目进展情况以及客户的需求的优先级。选择这个方法的理由是，客户需求的优先级和需求经常改变，所以制订能适应这些变化的灵活的计划是合理的。Cohn 的书 (Cohn 2005) 对敏捷计划进行了很好的介绍。

敏捷开发方法，例如 Scrum (Rubin 2003) 和极限编程 (Beck and Andres 2004)，使用的是一个两阶段的计划方法，对应于计划驱动的开发中的启动阶段和开发计划阶段。

1. 发布计划，包括对未来几月的展望以及决定系统的发布版本中应该包含的功能。

2. 迭代计划，包括短期的展望，主要是计划系统的下一个增量。一般这将花费整个团队 2~4 周的工作量。

第 3 章讨论了 Scrum 计划方法，它基于要完成的工作的项目待办事项和每日评审。它主要适合迭代计划。另一个敏捷计划的方法是基于用户故事，是作为极限编程的一部分提出来的。所谓的计划游戏可以用于发布计划和迭代计划。

计划游戏 (见图 23-8) 的基础是一组用户故事，这些用户故事 (见第 3 章) 能覆盖最终系统中包括的所有功能。开发团队和软件客户一起工作来确定这些故事。开发团队成员阅读并讨论这些故事，然后按照实现这些故事所需要的时间将故事排序。有些故事可能较大，以至于一次迭代不能够实现，因此这些较大的故事会被拆分成较小的故事。



图 23-8 计划游戏

将故事排序的难点在于，人们经常发现评估做某些事情所需要的工作量和时间是很困难的。为了简化该过程，需要使用相对排序。不用精确地估计需要付出的工作量，团队成对地比较这些故事并确定哪些故事需要最多的工作量和时间，从而不需要确切地评估每个场景需要的工作量。在该过程的最后，用户故事的列表是有序的，并且列表中最前面的故事实现时需要付出最多的工作量。随后，项目组给列表中的这些故事分配理论工作量点。一个复杂的故事需要 8 个点，简单的需要 2 个点。

进行了故事估算之后，使用一个“速度”的概念，将相对工作量转变成对所要求的总体工作量的第一次估计。速度是项目组每天所能完成的工作量点数量。这可以通过之前的经验估计，或者通过实现一两个故事来了解需要多少时间。速度估计虽然是近似的，但是可以在开发过程中进一步精练。一旦有了速度估计，就能够以人日为单位计算实现整个系统所需的工作量。

发布计划包括选择和完善上述故事，这些故事反映了在系统的发布版本中应实现的功能以及实现这些故事的顺序。客户应该参与这个过程。接下来选择一个发布日期，检查故事以判断工作量估计是否满足发布日期。假如不满足的话，增加或者删除清单上的一些故事。

迭代计划是开发增量交付系统的第一步。选择迭代过程要实现的故事，故事的个数反映了交付一个可行系统的时间 (通常为 2~3 周) 和项目组的速度。迭代交付日期到达之时，即使并不是所有故事都已实现，这次开发迭代也宣告完成。项目组考虑已实现的故事，增加其工作量点。重新计算速度，将其用于下一个系统版本的计划。

每次开发迭代的开始,会有一个任务计划阶段,开发人员将故事拆分成各个开发任务。一项开发任务大概花费4~16h。列出所有这次迭代中必须完成的任务。每个开发者申请他们要完成的任务。每个开发者知道自己的开发速度,不能申请比他们在规定时间内能完成的更多的任务量。

这个任务分配方法有如下两个主要的好处:

1. 整个项目组对迭代过程中要完成的任务有一个整体认识。因此他们能够理解项目组其他成员的工作内容以及确定任务依赖关系后应和谁交流。

2. 每个开发者选择要完成的任务,并不是简单地由项目管理者分配任务。这样开发者对自己选择的任务有一种拥有感,这会激发他们更好地完成任务。

迭代过程进行到一半的时候,进行进展评审。这时,应该已完成一半的故事工作量点。所以,如果一次迭代包含24个故事点和36个任务。应该已完成12个故事点和18个任务。如果没有完成的话,必须征询客户的意见删除迭代中的一些故事。

这种计划方法有一个好处,软件增量一直在每次项目迭代的结尾进行交付。假如增量中包含的特征在允许的时间内不能完成,就将减少相应的工作量。交付进度任何时候都不会延长。但是,这意味着客户计划会受到影响从而产生问题。减量会给客户带来额外负担,因为他们不得不使用不完整的系统,或者要在两个系统版本之间的切换从而改变了他们的工作方式。

敏捷方法中的主要困难在于依赖于客户参与。客户参与很难进行安排,因为客户代表有时要先忙于其他工作并且可能没有时间进行计划游戏。客户可能更加熟悉传统项目计划,也许很难参与到敏捷计划项目中。

对于那些能聚在一起讨论要实现的故事的小型、稳定的开发团队而言,敏捷方法能够很好地运作。但是对于那些庞大并且分布在不同地点的项目组或者组员频繁变动的项目组而言,实际上不可能每个人参与到敏捷项目管理中最为核心的集体计划中。所以,通常使用传统软件管理方法对大型项目进行计划。

23.5 估算技术

项目进度估算非常困难。初始的估算可能需要根据不完整的用户需求定义做出。软件可能需要运行于某些特殊类型的平台上,或者需要运用到新的开发技术。项目管理者对参与到项目中来的人员的技术水平可能还一无所知。如此多的不确定因素意味着,在项目早期阶段对系统开发成本进行精确估算是相当困难的。尽管如此,组织还是需要对软件所需工作量和成本进行估算的。有以下两种类型的估算技术。

1. 基于经验的技术。使用管理者之前项目和应用领域的经验估算要求的未来工作量,即管理者主观给出所需要的工作量的一个估计。

2. 算法成本建模。这类方法使用一种公式化的方法计算项目的工作量,基于对产品属性(规模、过程特点和参与员工的经验)的估计。

无论以上哪种技术,都需要使用判断力直接估算工作量或者估算项目和产品特点。在项目的启动阶段,估计的偏差比较大。基于从大量项目中收集的数据,Boehm等(B. Boehm et al. 1995)发现启动阶段的估算差异巨大。假如开始的工作量估计是 x 个月,那么系统交付时测量的实际工作量范围可能是 $0.25x \sim 4x$ 。在开发计划中,随着项目的进行估算会越来越准确(见图23-9)。

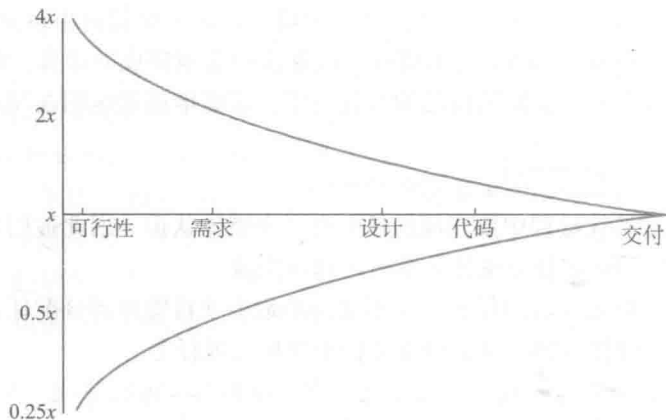


图 23-9 估算的不确定性

基于经验的技术依赖管理者之前项目的经验以及这些项目中有关软件开发活动上的实际工作量投入。通常，项目管理者需要定义项目中要生成的可交付物以及不同的软件构件或者要开发的系统。将这些内容记录在电子表格中，单独估算各个子项，再计算需要的总工作量。通常召集一组人参与估算工作量，并要求每个组员解释各自的估算结果，这样做很有帮助。很多情况下这将暴露别人没考虑到的因素，管理者重复这个过程谋求一个达成共识的估算结果。

基于经验的技术的困难在于一项新软件项目可能和之前的项目没有太多的共同点。软件开发变化非常快，软件开发项目经常使用一些不熟悉的技术，例如 Web 服务、应用系统配置、HTML5。如果管理者没有使用过这些技术，之前的经验可能对估算需要的工作量没有帮助，从而使准确的成本和进度估算更加困难。

很难说基于经验的方法和算法评估方法哪一个更加准确。项目估算通常是先入为主的。项目估算是用来确定项目预算，然后通过调整产品以保证预算不被突破。为了让项目开支控制在预算之内，可能会牺牲一些待开发的软件特性。

为了比较不同方法的准确性，需要进行大量的对照实验，每一种方法均独立地用来估算项目的工作量和成本。在实验的过程中项目不允许被修改，因此，最终由每种方法估算的工作量就可以进行比较。项目管理者不会知道工作量估算，这样就可以保证不引入偏见。然而，这样的场景在实际项目中是完全不可能的，所以我们永远无法对这些方法进行客观地比较。

23.5.1 算法成本建模

算法成本建模基于对项目的规模的估算、开发的项目类型、其他团队、过程和产品因素，用一个数学公式预测项目的成本。算法成本模型可以通过对已完成项目的成本及其特性的分析建立起来，并找到最接近的公式适应实际情况。

算法成本模型主要用于估算软件的开发成本，但 Boehm 以及他的合作者 (B. W. Boehm et al. 2000) 讨论了这些模型的另外一些用途，例如为软件公司投资者做估算准备，帮助评估风险的可选策略，以及关于经验性的复用、再开发或外购的决策。

软件项目中的工作量估算的算法模型可基于一个简单的公式：

$$\text{Effort} = A \times \text{Size}^B \times M$$

A, 一个常量因子, 依赖于组织的实践经验和所开发的软件类型。

Size, 一个对软件的代码规模或是用功能点或应用点表示的功能的估算。

B, 表示软件复杂度, 其值通常在为 $1 \sim 1.5$ 。

M, 一个乘数因子, 反映了过程、产品、开发属性, 例如软件的可依赖性需求, 以及开发团队的经验等综合因素。这些属性有可能造成开发系统总体难度的增加或降低。

交付的系统中的源代码的行数 (Source Lines Of Code, SLOC) 是基本的规模度量标准, 可以用于许多的算法成本建模。为了估计系统中代码行数, 我们可以组合使用以下的方法:

1. 对比待开发的系统与已开发的相似系统, 并基于相似系统的代码规模进行估算。
2. 估算系统中的功能点和应用点的数量 (见前面的章节), 并公式化地将其转化为相应编程语言所对应的代码行数。
3. 根据程序中构件的相对大小将其排序, 然后通过已知的参考构件来将之前排序好的构件转化成代码规模。

绝大多数算法估算模型都有指数成分 (上式中的 B), 这与规模和系统的复杂度成正比。这反映了一个事实, 即成本一般都不是与项目规模呈线性关系的。随着软件规模和复杂度的增大, 扩大的团队的通信费用在增加, 需要的配置管理更复杂, 系统的集成难度也在加大, 所有这些都要付出额外的费用。系统越复杂, 这些因素影响成本越多。

用科学客观的方法进行成本估算的想法很有吸引力, 但是所有的估算模型都存在以下两个主要问题:

1. 在项目早期阶段只有规格说明存在的情况下, 估算 Size 通常是个难题。功能点和应用点估算 (后面将谈到) 比估算代码长度要容易, 但是不够精确。
2. 因子 B 和 M 的估算往往带有主观色彩, 这样主观估算的结果往往因人而异, 取决于个人的环境背景和对于正在开发项目类型的经验。

准确的代码规模估算在项目早期是很难的, 因为最终程序规模依赖于设计决策, 而需要估算的时候决策还未形成。举例来说, 一个需要高性能复杂数据管理的应用, 可以使用自己实现的数据管理工具, 也可以使用商业数据库。在最初的成本估计时, 不可能知道是否存在表现足够好、能够满足运行需求的商业数据库系统。所以不知道要在系统中加入多少数据管理代码。

系统开发所使用的程序语言也会产生重要影响。使用像 Java 这样的语言较之使用 C 语言, 可能意味着需要较多的程序代码。不过, 这些额外的代码会带来更多编译时检查, 所以检验成本就会降低。估算过程中如何考虑这些因素目前尚不明了。此外, 代码复用的估算方法也不尽相同, 目前有一些模型可以明确地估算复用的代码行数。然而, 如果拥有应用系统或者外部服务的复用, 那么通常很难计算源代码中被替换的代码行数。

算法成本模型是估算开发一套系统所需工作量的系统方法。然而, 这些模型很复杂而且很难使用。模型中有很多的属性, 在估算它们的值的时候有相当大的不确定性。这种复杂性意味着算法成本建模的实际应用只限于一小部分大型公司, 这些公司大多涉及国防和航空航天系统工程工作。

使用算法模型的另一个障碍是需要校准。模型用户应该根据他们自己的历史项目数据校准他们的模型和参数值, 因为这反映了部门实际和经验。但是几乎没有组织从过去的项目中收集了足够的数据, 这种数据会以一种表格的形式存在以支持模型校准。因此, 算法模型的实际使用必须以模型参数的发布值开始。实际上建模者不可能知道这些是如何紧密地关联到

他们自己的组织的。

如果算法模型用于项目成本计算，估算者应该做一系列估算（最坏值、期望值和最好值）而不是单一估算，并用成本计算公式都计算一遍。只有在对开发的软件类型非常了解，所使用的模型经过组织的长期使用，其参数已经校正得较为准确，而且语言和硬件选择都预先确定了的情况下，成本估算才最有可能取得精确的结果。



软件生产率

软件生产率是对软件工程师在每周或每个月完成的开发工作的一个平均数量的估计。因而它表示为每月代码行数、每月功能点数等。

然而，在具有有形的结果（例如，办事员每天处理 N 个差旅费报销单）的地方，生产率是容易计算出来的，而软件生产率相对而言就十分难于定义。不同的人对于相同的功能会用不同的方式实现，结果他们使用的代码行数就会不同。代码的质量也是重要的，但是在某种程度上，是主观性的。因此，比较不同软件工程师间的生产率是很不可靠的。生产率对于较大的团队而言才是有意义的。

<http://software-engineering-book.com/web/productivity/>

23.6 COCOMO 成本建模

目前最著名的算法性成本建模方法和工具是 COCOMO II 模型。COCOMO II 模型完全是一个经验模型。获得该模型的途径是这样的：首先从大量的大小不同的软件项目中收集数据，然后通过对这些数据的分析找出与观察资料最相符合的公式。这些公式将系统的规模、产品、项目和团队因素等与开发系统的工作量联系起来。COCOMO II 是一个由开源工具支持的免费模型。

COCOMO II 是从更早期的 COCOMO (Constructive Cost Modeling) 成本估算模型基础上发展而成的，早期的 COCOMO 模型基本上是基于原始的代码开发的 (B. W. Boehm 1981; B. Boehm and Royce 1989)。COCOMO II 模型考虑了现代的软件开发方法，例如使用动态语言的快速开发、基于复用的开发、数据库编程等。COCOMO II 包含几个基于这些技术的子模型，这些子模型可以产生越来越详细的估算。

作为 COCOMO II 模型一部分的子模型（见图 23-10）包括以下这些：

1. 应用组合模型。它对所需的工作量建模，所开发的系统是由可复用构件、脚本或数据库编程创建而得。软件规模估算基于应用点，还可根据一个简单的规模 / 生产率公式来估算所需的工作量。

2. 早期设计模型。这个模型用于在得到需求之后的早期系统设计阶段。估算是基于在本章开头部分谈到的标准估计公式，带有包含 7 个因子的简化的参数。使用功能点进行估算，然后转化为源代码的行数。

功能点是一种独立于语言的量化程序功能的方式。通过衡量或者估算外部输入和输出的数量、用户迭代的次数、外部界面的数量，以及系统使用的文件或数据库的数量，计算出程序中的所有功能点数目。

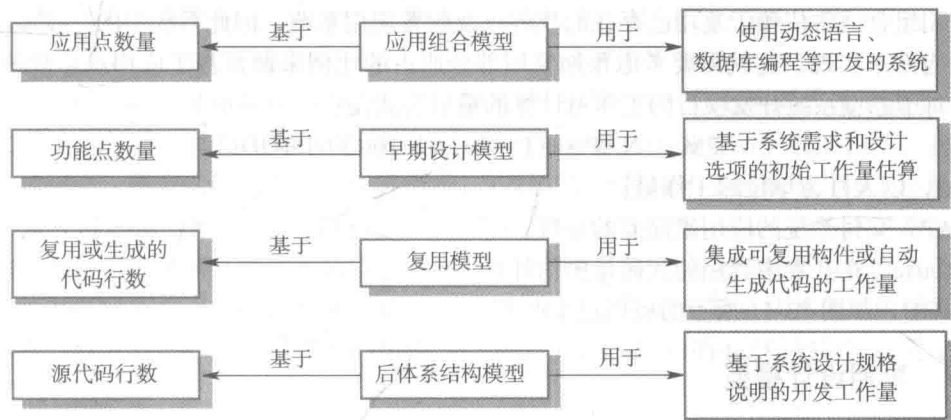


图 23-10 COCOMO 估算模型

3. 复用模型。这个模型用于计算集成可复用构件和自动生成程序代码所需的工作量。它一般与后体系结构模型结合使用。

4. 后体系结构模型。当设计出系统体系结构后，就可以对软件规模做更精确的估算。这个模型也使用上面谈到的成本估算的标准公式，但它包含了更广泛的有 17 个反映个人能力和产品与项目特征的参数集合。

当然，在大型系统中，不同部分可能使用不同的技术进行开发，不必要求以同等的精确度估算系统的所有部分。在这种情况下可以为每个部分采用适当的子模型，然后合并这些结果得到一个合成的估算值。

COCOMO II 是一个非常复杂的模型，为了更加简便地描述，本文简化了它的表示。你可以使用此处解释的模型来做一些简单的成本估算。然而，如果想要真正合理地使用 COCOMO II，则需要参考 Boehm 的书籍以及 COCOMO II 的指南 (B. W. Boehm et al. 2000; Abts et al. 2000)。

23.6.1 应用组合模型

将应用组合模型引入 COCOMO II 以支持对项目所需要的工作量进行估算。此模型适合于原型构造型项目和通过已有的构件组合进行软件开发的项目。它的计算是基于加权的应用点 (有时也称为对象点) 除以一个标准应用点生产率 (B. W. Boehm et al. 2000)。一个程序中的应用点数量的估算由下面 4 个更加简单的估算导出。

- 显示的单独的屏幕和网页的数量；
- 显示的产生报表的数量；
- 命令式程序设计语言 (如 Java) 中模块的数量；
- 脚本语言或者数据库编程的代码行数。

然后按照开发每个应用点的难度对估算值进行调整。程序员的生产率取决于开发者的经验、能力以及所使用的软件工具 (ICASE) 的水平。图 23-11 是 COCOMO 模型开发者给出的不同应用点生产率水平。

开发者的经验和能力	非常低	低	一般	高	非常高
软件工具的成熟度和能力	非常低	低	一般	高	非常高
PROD (NOP/月)	4	7	13	25	50

图 23-11 应用点生产率

应用组合通常依赖于复用已存在的软件以及配置应用系统,因此系统中的一些应用点由可复用构件来实现。这就需要考虑预期复用部分所占的比例来调整基于应用点总数的估算。因此,对于原型系统开发项目的工作量计算的最后公式是:

$$PM = (NAP \times (1 - \%reuse/100)) / PROD$$

PM, 以人月为单位的工作量;

NAP, 交付系统的应用程序点的总数;

%reuse, 在开发中利用的代码量的估计;

PROD, 如图 23-11 所示的应用点生产率。

23.6.2 早期设计模型

在项目的初始阶段使用这个模型,此时还未对系统体系结构做出详细的设计。早期设计模型假定用户需求已经确定,系统设计过程的初始阶段已经开始。这一阶段的任务应当是简单快速地做一个大略的成本估算。因此需要做出各种简单化的假设,例如集成可复用代码的工作量为零。

早期设计估算对于选择探索是最为有用的,此时我们需要比较实现用户需求的各种方式。在这一阶段做出的估算是基于算法模型的标准公式得出的,即

$$Effort = A \times Size^B \times M$$

Boehm 基于他自己的大型数据集提出对于这一阶段的估算,系数 A 应该为 2.94,系统的规模用 KSLOC 表示,即源程序以千行代码为单位的数量。这是通过估算软件中功能点数,然后使用标准表格将功能点转换成 KSLOC 来计算的,这个标准表格针对不同语言(QSM 2014),给出功能点和软件规模的对应值。

指数 B 反映了随着项目规模的扩大所需工作量的增长。它可以在 1.1 ~ 1.24 之间变动,与项目的创新程度、开发的灵活性、采用的风险处理过程、开发团队凝聚力,以及组织的过程成熟度(见第 26 章)水平等密切相关。在关于 COCOMO II 的后体系结构模型中会描述用这些参数计算该指数的方法。

由此得到下列工作量计算公式:

$$PM = 2.94 \times Size^{(1.1 \sim 1.24)} \times M$$

$$M = PERS \times PREX \times RCPX \times RUSE \times PDIF \times SCED \times FSIL$$

PERS, 个人能力;

PREX, 个人经验;

RCPX, 产品可靠性和复杂性;

RUSE, 所要求的复用;

PDIF, 平台困难程度;

SCED, 进度;

FSIL, 支持设施。

乘数因子 M 基于 7 个项目和过程属性,这可以增加或减少估算值。本书的网页上有对这些属性的解释。对这些因子的取值可以分成 6 档,“非常低”的值该因子赋值 1,对具有“非常高”的值该因子赋值为 6。例如 PERS=6,说明这个专家员工可以参与到项目中。

23.6.3 复用模型

复用模型用于估计集成可复用代码或已生成代码所需的工作量。如在第 15 章所述,软

件复用在软件开发中很普遍。大多数大型系统都有很大一部分代码是从以前开发的系统中复用而来的。

COCOMO II 将复用的代码分为两类。黑盒代码是那种不需去理解或做出改动的代码。黑盒代码的例子是那些由 UML 模型或者应用库（如图形库）里自动生成的构件。我们假设在开发过程中，黑盒代码的开发工作量为零，并且它的规模也不会影响到总体工作量中。

白盒代码需要将新代码或其他复用构件的代码改编整合在一起。它的复用需要一些开发工作，因为需要理解并修改代码以使它在系统中正常工作。白盒代码也可以是自动生成的代码，但是其代码需要人工修改或者添加。另外，白盒代码也可能是从其他系统中复用过来的需要在新开发的系统中进行修改的构件。以下是 3 个影响复用白盒代码构件所需工作量的因素：

1. 评估某个构件是否可以在开发系统中复用所需的工作量。
2. 理解复用代码所需的工作量。
3. 修改复用代码并集成到开发系统中所需的工作量。

我们通过 COCOMO 的早期设计模型来计算复用模型的开发工作量，这样的计算是基于系统的总代码行数的。代码规模包括开发不能复用的构件所写的新代码的工作量以及复用和集成已存在代码的额外工作量。这样的额外工作量叫作 ESLOC，它相当于新的源代码的行数。也就是说，复用工作量作为开发新的额外源代码的工作量。

计算这种等价关系的公式为：

$$ESLOC = (ASLOC \times (1 - AT/100) \times AAM)$$

ESLOC，新源代码的等价行数；

ASLOC，必须修改的复用构件的代码行数；

AT，可以自动修改的复用代码所占的百分比；

AAM，改写调整因子，反映了构件复用时所需的额外工作量。

在某些情况下，复用代码需要对语法进行修改，这时可以通过自动化工具来实现。用自动化工具实现并不需要太多工作量，因此，需要估算在复用代码中有多少修改是可以进行自动化修改的（AT）。这就减少了所需修改的代码总行数。

改写调整因子（Adaptation Adjustment Multiplier, AAM）调整估算以反映复用代码需要的额外的工作量。COCOMO 模型文档（Abts et al. 2000）中详细阐述了 AAM 的计算。简单来说，AAM 是以下 3 个部分之和：

1. 评估因子（assessment factor, AA），表示决策构件是否能够进行复用的工作量。根据寻求并评估潜在候选复用所需的时间，AA 的值为 0 ~ 8。
2. 理解部分（understanding component, SU），表示理解要复用的代码以及工程师熟悉这些代码所需的工作量。SU 的范围为 50 ~ 10，50 表示复杂的非结构化代码，10 表示书写良好的面向对象的代码。
3. 改写部分（AAF），表示修改这些复用代码的成本。改写部分包括设计、编码和集成变更等几个子部分。

一旦计算出 ESLOC，就可以应用标准估算公式计算总工作量，参数 Size 等于 ESLOC。因此估算复用工作量的公式为：

$$Effort = A \times ESLOC^B \times M$$

式中 A、B 和 M 的含义与早期设计模型中的相同。



COCOMO 成本驱动因素

COCOMO II 成本驱动因素是反映产品、团队、过程以及组织因素的某些属性，这些属性影响软件系统开发中所需的工作量。例如，如果需要高级别的可靠性，那就需发生要投入额外的工作量；如果有快速交付的需要，那同样会有额外的工作量；如果团队成员发生变化，也会需要额外的工作量。

在 COCOMO II 模型中有 17 个这样的属性，此模型的提出者给这些属性赋予了特定的值。

<http://software-engineering-book.com/web/cost-drivers/>

23.6.4 后体系结构模型

后体系结构模型是 COCOMO II 模型中最详细的一个。当已经有了系统的初始体系结构设计之后使用该模型。在后体系结构模型产生的估算所依据的基本公式与早期设计模型估算中的基本公式相同：

$$PM = A \times Size^B \times M$$

到这个阶段为止，因为知道如何将系统分解为各个子系统和构件，软件规模的估算在这个阶段中要准确得多。估算代码规模需要下面 3 个参数：

- 1. 要开发的新代码行数的估算 (SLOC)；
- 2. 基于复用模型计算得到的等价代码行数 (ESLOC) 的复用成本的估算；
- 3. 由于需求变更可能要修改的代码行数的估算。

最后一个估算值——要修改的代码行数——反映了软件需求经常会变化。应该考虑这将引起重复工作以及开发额外的代码。当然这个数值常常比要开发的新代码的估算有更多的不确定性。

在工作量计算公式中的指数项与项目的复杂程度有关。当项目较复杂时，增加系统规模所花费的工作量明显增大。通过分析如图 23-12 所示的 5 个因子来计算指数 B 的值。这些因子都有从 0 ~ 5 六个等级，0 表示“特别高”，5 表示“非常低”。计算 B 时，将这些估算值相加再除以 100，然后再加上 1.01 就是该指数项的取值了。

规模因子	解 释
体系结构 / 风险解决方案	反映了所执行的风险分析的程度。非常低意味着几乎没有分析；非常高意味着完全彻底的风险分析
开发灵活性	反映了开发过程的灵活性级别。非常低意味着使用一个预先指定的过程；非常高意味着客户只设定了总目标
有先例可循	反映了组织在此类型项目上先前所获得的经验。非常低意味着没有先前经验；非常高意味着组织非常熟悉此应用
团队凝聚力	反映了开发团队成员彼此了解和一起和谐工作的程度。非常低意味着交互存在很大困难；非常高意味着一个团结高效的团队，没有沟通障碍
过程成熟度	反映了组织的过程成熟度（如在线的第 26 章中所讨论的）。该值的计算依赖于 CMM 成熟度调查问卷，但是可以通过从 5 中减去 CMM 过程成熟度级别来获得一个估计

图 23-12 在后体系结构模型指数计算中所用的规模因子

例如,假定一个组织正承担一个项目,组织对于该项目所在领域没有经验。项目客户没有定义需要采用的过程,在项目进度中也没有安排重大风险分析,而且还需要组织一个新的开发团队来完成这个系统。该组织最近刚实行过程改善计划,并且依据 SEI 能力评估模型,如第 26 章(在线章节)中讨论的,已经被评定为 2 级组织。在进行指数计算时用于评级的可能取值如下。

1. 有先例可循,取值为“低”(4)。这是组织的一个新项目。

2. 开发的灵活性,取值为“非常高”(1)。开发过程没有客户介入,所以几乎不存在外部强加的改变。

3. 体系结构/风险解决方案,取值为“非常低”(5)。没有风险分析。

4. 团队凝聚力,取值为“一般”(3)。是个新团队,没有相关信息。

5. 过程成熟度,取值为“一般”(3)。有一些过程控制。

这些值的总和为 16。然后除以 100 计算指数,结果 0.16 加上 1.01,所以得到调整后的指数是 1.17。

总的工作量估算使用一套广泛的包含 17 个产品、过程、组织属性(见方框中的 COCOMO 成本驱动因素)而不是早期设计模型中所使用的 7 个属性进行精化。你可以估算这些属性的值,因为你有了更多的关于软件自身、它的非功能性需求、开发团队以及开发过程的信息。

图 23-13 给出一个例子,说明这些成本驱动因素属性是如何影响工作量估算的。在这里对指数所取的值为 1.17,与上一个例子相同。假设指标可靠性(RELY)、复杂度(CPLX)、存储(STOR)、工具(TOOL)和进度(SCED)是项目中的关键性成本驱动因素。所有其他成本驱动因素取标称值 1,不会影响工作量计算。

在图 23-13 中,对关键性的成本驱动因素分别赋予了最大和最小值来说明它们对工作量估算带来的影响。所取的值来自 COCOMO II 的参考手册(Abts et al. 2000)。从图中可以看出,对成本驱动因素取较高值时的工作量估算是初始估算的 3 倍多,而对这些成本驱动因素取较低值时的工作量估算会降低到初始估算的大约 1/3,这样就突出表示了不同类型项目之间的巨大差异,以及在将一个领域的经验移植到另一个领域时的巨大困难。

指数值	17
系统规模(包括了复用和需求易变性因素)	128 千行代码(KLOC)
不考虑成本驱动因素的初始 COCOMO 估计	730 人月
可靠性	非常高,乘数 = 1.39
复杂度	非常高,乘数 = 1.3
内存限制	高,乘数 = 1.21
工具使用	低,乘数 = 1.12
进度	加速的,乘数 = 1.29
调整的 COCOMO 估计	2306 人月
可靠性	非常低,乘数 = 0.75
复杂度	非常低,乘数 = 0.75
内存限制	无,乘数 = 1
工具使用	非常高,乘数 = 0.72
进度	正常,乘数 = 1
调整的 COCOMO 估计	295 人月

图 23-13 成本驱动因素对工作量估算的影响

23.6.5 项目的工期和人员配备

项目管理者除了要对软件系统开发所需成本及其工作量做出估算以外,还必须估算软件的开发周期以及人员要在什么时间到位。组织不断地想缩短开发进度,希望自己的产品能赶在竞争对手之前投放市场。

COCOMO 模型包括一个对项目所需日历时间的计算公式:

$$TDEV = 3 \times (PM)^{(0.33 + 0.2 * (B - 1.01))}$$

TDEV, 项目的理论进度 (月), 忽略关于项目进度的任何乘数;

PM, 使用 COCOMO 模型计算的工作量;

B, 在 23.5.2 节讨论的复杂度相关的指数。

如果 $B = 1.17$, $PM = 60$, 那么

$$TDEV = 3 \times (60)^{0.36} = 13 \text{ (月)}$$

然而, 使用 COCOMO 模型预测的理论工程进度和软件客户要求的进度不一定是一回事。可能需要比理论进度显示的日期提前或者延迟 (比较少见) 交付软件。如果时间进度被压缩 (比如软件被更快地开发了), 这会增加项目需要的工作量。工作量估算中的 SCED 乘数能解决此问题。

如上所述, 假定项目估计 TDEV 是 13 个月, 但是实际进度要求是 10 个月。这相当于将进度压缩大约 25%。使用由 Boehm 小组得到的 SCED 乘数的值, 这种进度压缩的工作量乘数是 1.43。因此, 如果加速后的进度能够实现, 实际交付软件需要的工作量多于理论进度要求的工作量的 50%。

项目所需的工作人数、需要的工作量以及项目的交付进度之间关系非常复杂。如果 4 人能在 13 个月内完成一个项目 (52 人月的工作量), 也许你会想到增加一个人, 那么就能在 11 个月内完成工作 (55 人月的工作量)。但是 COCOMO 模型显示, 实际上需要 6 个人才能在 11 个月内完成工作 (66 人月工作量)。

出现这种情况的原因是增加人员实际上降低了原有组员的生产率。当项目组规模增加的时候, 组员花费更多的时间交流以及定义由别人开发的系统各个部分的接口。人员数量翻一番 (打个比方), 项目工期并不是简单地缩减一半。

因此当你为团队添加一个新成员时, 实际带来的工作量收益通常是小于一个人的, 因为团队里的其他成员会变得低效。如果开发团队规模很大, 由于总体工作量效率的影响, 有些时候情况是增加项目人员增加了开发日程而不是减少了日程。

单纯用项目所需工作量除以开发所需时间, 对项目团队所需人数进行估算并没有什么帮助。一般来讲, 在项目刚开始的时候只需要很少几个人, 他们负责项目的初始设计。在项目开发和系统测试的时候, 项目成员数达到最高峰。当系统完成准备部署的时候, 人员数目开始下降。这就显示了项目人员的组织和项目时间的减少的关联关系。因此, 项目管理者在项目整个生存期中都要避免过早把太多人员加到项目中来。

要点

- 系统报价并不仅仅取决于系统开发成本估算和开发公司要求的利润。组织因素可能提高售价以补偿升高的风险, 或者降低售价以获得竞争的优势。
- 软件通常是先有定价以得到合同, 然后再据此调整相应的功能。

- 计划驱动开发是围绕一个详细定义的项目计划进行组织的。项目计划定义了项目活动、计划的工作量、活动进度安排和每项活动的负责人。
- 项目进度安排需要创建有关项目计划的各种图形化表示。用来表示活动持续时间和人员使用时间的条状图是在进度安排中最常使用的。
- 项目里程碑是一个项目活动可以预期的结果，到达一个里程碑就要把某些项目进展报告提交到管理层。在一个软件项目中，里程碑的出现应该是有规律的。可交付物则是交付到客户手中的工作产品。
- 敏捷计划游戏是让整个团队成员都参与到项目计划中来。计划做成增量式的，如果问题出现，计划将被调整，通过减少软件的功能性而不是延期交付一个增量。
- 软件的估算技术可能是基于经验的，管理者对需要的工作量进行判断；或者是使用算法，需要的工作量通过使用其他已估算工程中的参数计算得到。
- COCOMO II 成本模型是一个比较成熟的成本估算模型，它将项目、产品、硬件以及人的因素都考虑在内。

阅读推荐

第22章中的阅读推荐材料也与本章相关。

《Ten unmyths of Project Estimation》是一篇谈项目估算中实际困难并挑战这一领域一些基本假定的实用文章。(P. Armour, Comm. ACM, 45 (11), November 2002) <http://dx.doi.org/10.1145/581571.581582>

《Agile Estimating and Planning》这本书全面介绍如在XP中使用的基于故事的计划，同样也阐述了在项目计划中使用敏捷方法的基本原理。它包括一个很好的关于项目计划问题的概括。(M. Cohn, 2005, Prentice-Hall)

《Achievements and Challenges in COCOMO-based Software Resource Estimation》这篇文章展示了COCOMO模型的发展过程及其产生的影响，讨论了多种变形。它也介绍了COCOMO方法可能的发展前景。(B. W. Boehm and R. Valeridi, IEEE Software, 25 (5), September/October 2008) <http://dx.doi.org/10.1109/MS.2008.133>.

《All About Agile ; Agile Planning》这个关于敏捷方法的网站包括一组很好的敏捷计划的文章，这些文章是由不同的作者撰写的。(2007—2012) <http://www.allaboutagile.com/category/agile-planning/>

《Project Management Knowhow: Project Planning》这个网站有一些关于通用的项目管理的有用文章。这些文章针对的是此前在这个领域没有经验的人。(P. Stoemmer, 2009-2014) http://www.project-management-knowhow.com/project_Planning.html

网站

本章的PPT: <http://software-engineering-book.com/slides/chap23/>

支持视频的链接: <http://software-engineering-book.com/videos/software-management/>

练习

- 23.1 在什么情况下公司可能会将软件系统的价格定得比成本估算加上正常利润高得多呢?
- 23.2 项目计划过程为什么是一个迭代的过程? 为什么在一个软件项目期间必须不断地对项目

目计划进行评审？

- 23.3 简要介绍软件项目计划中每个部分的目的。
- 23.4 无论使用什么估算方法，成本估算有其固有的风险。试给出 4 个能降低成本估算风险的方法。
- 23.5 图 23-14 列出了许多活动、持续时间和各活动之间的依赖关系。请画出活动图和条状图来示意项目进度安排。

任务	持续时间 (天)	依赖关系
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

图 23-14 进度安排例子

- 23.6 图 23-14 给出了软件项目活动的任务持续时间。假设发生了一个严重的、意想不到的事件，使得任务 T5 不是在 10 天内完成，而是用了 40 天。画出新的条状图以表示如何重新进行组织。
- 23.7 计划游戏是基于计划的思想去实现表示系统需求的故事。解释当软件有高性能和高可依赖性需求时这个方法可能存在的问题。
- 23.8 有一个软件管理者负责一个安全性关键软件系统的开发，该系统的设计是为了控制针对癌症患者的放射治疗仪。这个系统是嵌入式系统，运行在一种专用处理器上，内存被限定在 256MB。放射治疗仪与患者数据库通信以获取患者的详细资料，并在治疗完毕后自动地将放射剂量和其他详细治疗信息记录到数据库中。
在对系统开发的工作量估算中使用了 COCOMO 方法，计算结果是需要 26 人月。在估算中所有的成本驱动因素都被设为 1。
解释为什么需要对这个估算进行修正，将项目、人员、产品和组织因素统统考虑在内。试列举出在初始 COCOMO 估算中会产生重要影响的 4 个因素，并对这些因素给出可能的取值。对于为什么考虑到这些因素给出合理的解释。
- 23.9 一些非常大型的软件项目都有数以百万行的程序代码行。解释为什么像 COCOMO 这样估算模型的工作量对非常大型的软件系统可能无效。

- 23.10 公司了解到客户需求不明确，在签订合同时故意提出一个低报价，待客户将来提出需求变更时再索要高价。你认为这样做道德吗？

参考文献

- Abts, C., B. Clark, S. Devnani-Chulani, and B. W. Boehm. 2000. "COCOMO II Model Definition Manual." Center for Software Engineering, University of Southern California. http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.o/CII_modelman2000.o.pdf
- Beck, K., and C. Andres. 2004. *Extreme Programming Explained: 2nd ed.* Boston: Addison-Wesley.
- Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. 1995. "Cost Models for Future Software Life Cycle Processes: COCOMO 2." *Annals of Software Engineering*: 1-31. doi:10.1007/BF02249046.
- Boehm, B., and W. Royce. 1989. "Ada COCOMO and the Ada Process Model." In *Proc. 5th COCOMO Users' Group Meeting*. Pittsburgh: Software Engineering Institute. <http://www.dtic.mil/dtic/tr/fulltext/u2/a243476.pdf>
- Boehm, B. W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.
- Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. 2000. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice-Hall.
- Cohn, M. 2005. *Agile Estimating and Planning*. Englewood-Cliffs, NJ: Prentice Hall.
- QSM. 2014. "Function Point Languages Table." <http://www.qsm.com/resources/function-point-languages-table>
- Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.

质量管理

目标

本章旨在介绍软件质量管理，讲述专门的质量管理活动。阅读完本章后，你将：

- 了解质量管理过程以及质量规划的重要性；
- 了解质量管理过程中质量标准的重要性以及标准是如何用于质量保证的；
- 了解评审和审查是如何作为机制在软件质量保证中使用的；
- 理解敏捷方法中的质量管理是如何基于团队质量文化的发展而进行的；
- 理解度量是如何在评估某些软件质量属性时发挥作用的，以及目前软件度量的局限性。

软件质量管理与确保开发的软件系统符合意图息息相关。换言之，软件系统不仅应该满足用户的需求，还应该高效、可靠地运行，同时要不超出预算并能够按时交付。在过去的 20 年里，质量管理技术、新型软件技术和测试方法的使用大大提高了软件质量的水平。

对于那些需要用几年时间开发大规模且生命周期长的系统的团队，正式的质量管理（Quality Management, QM）显得尤为重要。这些系统的用户是外部客户，通常用的是基于计划的开发过程。对于这些系统，质量管理既是组织层面的重要问题，也是单个项目层面的重要问题。

1. 在组织层面，质量管理与建立能生产高质量软件的组织过程框架和标准相关。这就意味着质量管理团队应该负责定义要使用的软件开发过程、软件应当符合的标准，包括系统需求、设计以及代码的相关文档。

2. 在项目层面，质量管理包括：专门的质量过程的应用，对所规划的过程执行情况的检查，以及确保项目的输出符合项目所适用的标准。项目层面的质量管理同样包括为项目确立一个质量计划。质量计划应该给出项目的质量目标，定义应该使用什么样的过程和标准。

软件质量管理技术源于制造业中的方法和技术，制造业中广泛使用质量保证和质量控制这两个术语。质量保证（Quality Assurance, QA）是对生产高质量的过程和标准的定义，同时也把质量过程引入制造过程。质量控制是应用这些质量过程淘汰没有达到质量要求的产品。质量保证和质量控制都是质量管理的一部分。

在软件产业中，一些公司认为质量保证仅是对旨在保证软件质量达标的规程、过程和标准的定义。而在另一些公司中，质量保证也包括开发团队交付产品后采取的活动，包括所有的配置管理、验证和确认活动。

质量管理对软件开发过程提供独立的检查。质量管理团队检验项目可交付成果，从而确保它们能够符合组织的标准和目标（见图 24-1）。质量管理团队还会检验软件过程文档，这些文档记录了项目中每个团队已经完成任务。质量管理团队使用文档来检验重要的任务是否被遗漏或者一个小组是否错误地假设其他小组已经完成了某个任务。

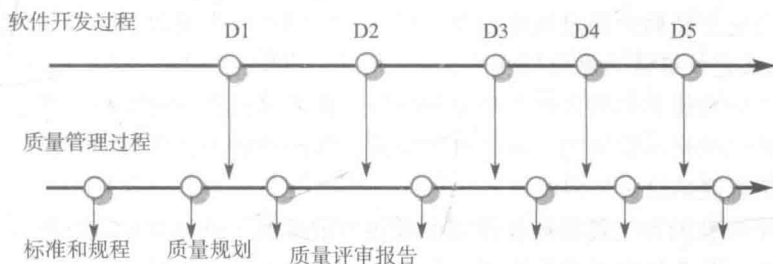


图 24-1 质量管理和软件开发

大型公司中的质量管理团队通常负责管理测试过程的发布。正如第 8 章中所讨论的，这就意味着在软件版本交付到客户手里之前，他们管理软件的测试。他们负责检查系统是否满足需求以及维护测试过程记录。

质量管理团队应该是独立于开发小组的团队，这样他们能够客观地对待软件产品的质量，能够不受软件开发问题的影响，做出客观的软件质量报告。理想情况下，质量管理团队在质量管理上负有组织范围内的责任，他们应该向高级管理层汇报。

因为项目管理人员必须维护工程预算开支和进度安排，所以他们可能在产品质量上做出妥协来满足工程进度。一个独立的质量管理团队确保组织的质量目标不会因为短期的预算和进度而受到影响。但是，在小一点的公司里，这是不太实际的。质量管理和软件开发不可避免地绑定在一起，相关的人员同时有着开发和质量两方面的职责。

正式的质量规划是基于计划的开发过程的组成部分。质量规划是为项目制定一个质量计划的过程。质量计划应当列出要达到的软件质量，并且描述怎样评估这些质量。对于一个特定系统，质量计划定义了何为高质量的软件。因此，工程师在最重要的软件质量属性上达成共识。

Humphrey (Humphrey 1989) 在他的关于软件管理的经典书籍中提出了一个质量规划的概要结构。其中包括：

1. 产品介绍，说明产品、产品的目标市场及对产品质量的预期。
2. 产品计划，包括产品的严格发布日期、产品责任以及产品的销售和售后服务计划。
3. 过程描述，产品的开发和管理中应该采用的开发、服务过程 and 标准。
4. 质量目标，产品的质量目标和计划，包括识别和判定产品的关键质量属性。
5. 风险和风险管理，说明影响产品质量的主要风险和这些风险的应对措施。

质量规划，作为一般项目规划过程的一部分，依据所开发的系统的大小和类型而有所不同。但是，在书写质量规划时，我们应该保证它们尽可能简短。如果文档过长，人们就不会去阅读，这样就会导致设定质量规划的初衷失败。

传统的质量管理是一个正式的过程，该过程包括维护大量有关测试和系统验证的文档以及如何遵循流程。在这方面，传统的质量管理与敏捷开发完全相反。敏捷开发的目的是在编写文档和正式确定如何进行开发工作上尽可能花费最少的时间。因此，当使用敏捷方法时，质量管理技术也需要随之演化。24.4 节将讨论质量管理与敏捷开发。

24.1 软件质量

质量管理的基本原则是制造工业为了改善制造产品的质量而建立的。作为质量管理的一部分，工业界首先定义了什么是“质量”，所谓质量是符合详细产品规格说明的。基本假设是，能够完整详细地定义产品，并且能够确立一个依照产品规格说明检查制造产品的规程。

当然，产品不会完全精确地满足规格说明，所以一定的误差是可以容忍的。如果产品是“基本合适的”，那么它就被认为是可接受的。

软件质量不能直接和制造业中的质量相比较。误差容忍思想对于模拟系统是适用的，但是对于软件系统来说是不适用的。由于以下原因，我们经常无法针对一个软件系统是否满足其规格说明得出客观结论：

- 1. 写出一个完整的和无歧义的软件需求是相当困难的。软件开发人员和客户可能对于需求有不同的解读，并且可能对软件是否符合规格说明没法达成共识。
- 2. 规格说明通常整合了各类利益相关者的需求。这些需求不可避免地存在着取舍，很可能没有包含所有利益相关者的需求。排除在外的利益相关者可能认为系统质量糟糕，即使它实现了那些达成共识的需求。
- 3. 对某些质量特性（如可维护性）的直接度量是不可能做到的，所以它们无法以一种无歧义的方式刻画。24.4 节讨论了度量的困难性。

因为以上这些原因，评估软件质量仍然是一个主观的过程，质量管理团队必须判断软件是否达到可接受的质量水平。质量管理团队必须考虑软件是否达到既定的目标。这涉及回答关于系统特性的若干问题。例如：

- 1. 软件是否得到了充分的测试，并且显示软件已经实现了所有需求？
- 2. 软件是否足够可靠能投入使用？
- 3. 软件性能是否对于正常使用是可接受的？
- 4. 软件是否可用？
- 5. 软件是否结构良好并且易于理解？
- 6. 开发过程是否遵循编程和文档化标准？

软件质量管理存在一个通用的假设：按照需求测试系统。应该根据这些测试的结果判断是否实现了要求的功能。因此，质量管理（QM）团队应该评审所设计的测试并检查测试记录，以核实测试是否被正确地执行。在一些公司中，软件管理团队负责最终的系统测试。但是有时是一个独立的系统测试小组负责向系统质量管理者汇报。

一个软件系统的主观质量很大部分依赖于它的非功能性特性。这反映了实际的用户体验——如果软件的功能不是所期望的那样，那么用户就会变通，寻找其他方式来做他们想做的事情。但是如果软件不可靠或者是速度太慢，那么实际上就不能达到他们的目的。

因此，软件质量不仅仅取决于软件功能是否正确地实现，也取决于非功能的系统属性，如图 24-2 所示。这些属性和软件可依赖性、可用性、效率以及可维护性相关。

安全性	可理解性	可移植性
信息安全性	可测试性	可用性
可靠性	可调节性	可复用性
韧性	模块化	效率
鲁棒性	复杂度	可学习性

图 24-2 软件质量属性

对于任何系统，优化所有属性都是不太可能的，例如，提升信息安全性可能导致性能的降低。因此质量规划应该定义被开发软件最重要的质量属性。可能效率很关键，需要牺牲其他属性来保证效率。如果已经在质量规划中强调了效率的重要性，从事开发工作的工程师会

协力来达到这一目标。质量规划还应该定义质量评估过程。该过程应当提供一种各方都认同的方式来确定一些质量属性（如可维护性和鲁棒性）是否在产品中得到了实现。

传统软件质量管理的一个基本假定是，软件开发过程的质量直接影响软件的质量。这个假定源于生产制造系统中产品质量与生产过程的密切关系。制造过程包括配置、安装和操作相应的机器。一旦机器操作正常，产品质量自然就有保证。你可以评估产品质量并改变生产过程，直到达到需要的质量水平。图 24-3 描述了这个基于过程实现产品质量的方法。

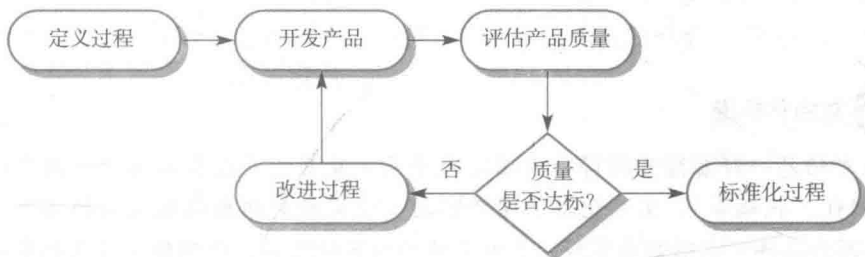


图 24-3 基于过程的质量

在制造业中过程与产品质量有着明确的关联，因为过程相对易于标准化和监控。一旦制造系统校准后就能一次次地生产出高质量的产品。软件不是生产制造出来的，而是设计出来的。因此，软件开发过程中过程质量和产品质量之间的关系更加复杂。软件开发是创造性活动，个人的技能和经验的影响非常大。无论所使用的过程怎样，一些外部因素（如应用程序的创新性或早期产品发布的商业压力）也会影响产品质量。

毫无疑问，使用的开发过程对于软件质量有明显的影响。好的过程更有可能得到高质量的软件。过程质量的管理和改进能够减少软件开发过程中产生的缺陷。但是，评估软件质量的属性非常困难，比如，不经过长时间使用软件，很难评估可维护性。因此，很难指出过程特性如何影响这些属性。此外，因为设计和创造性在软件过程中所起的作用，过程标准化有时会扼杀创造力，这会导致软件质量更糟而不是更好。

定义过程固然是很重要的，但是质量管理者也应该致力于推行一种“质量文化”，让每个参与产品开发的人都有强烈的产品质量意识。质量管理者应该鼓励团队成员对自己的工作质量负责，鼓励他们探求改善质量的新方法。质量标准和规程是质量管理的基础，同时好的质量管理者也认识到有些软件质量特性是无形的（如简洁性、可读性等），难以在标准中具体体现出来。他们应该支持那些关注无形质量的人，并鼓励所有的团队成员拿出好的工作作风。

24.2 软件标准

软件标准在基于计划的软件质量管理中扮演着重要的角色。正如前文所述，质量保证的一个重要部分是定义和选择应用于软件开发过程和软件产品的标准。作为质量保证过程的一部分，也要选择支持标准使用的工具和方法。一旦选定使用的标准，必须定义项目特定的过程以监控标准的使用和执行情况。

软件标准非常重要，有以下 3 个原因。

1. 标准是智慧的结晶，对一个组织有重要意义。软件标准封装了对于组织来说最成功的或是最适合的软件开发实践。这些知识往往是经过反复实验和无数的挫折后才得出的。把这

- 些知识制定到标准中去可以帮助企业复用以往的经验并避免重犯过去的错误。
2. 标准为定义特定环境中的“质量”提供了一个框架。如前文所述，软件质量是主观的，标准的使用为判断软件是否达到要求的质量水平提供了基础。当然，这取决于所设定的标准是否反映了用户对软件可依赖性、可用性以及性能的期望。
3. 软件标准还有助于工作的延续性，由一个人着手进行的工作别人可以接着做。软件标准确保一个组织中所有的工程人员采用相同的做法。这样一来，开始一项新工作时就节省了学习时间。



文档化标准

项目文档是一种看得见摸得着的描述软件系统及其生产过程的各种不同表示的方式（需求、UML、代码等）。文档化标准定义不同类型文档的组成以及文档的格式。这是很重要的，因为这样可以很容易发现是否有重要的内容被遗漏，并确保项目文档有一个普遍接受的外观。标准会针对书写文档的过程、文档本身的内容以及文档交换诸多方面分别制定。

<http://software-engineering-book.com/web/documentation-standards/>

在软件质量管理中，有两类可用于定义和使用的相关软件工程标准。

1. 产品标准。这些标准用于开发的软件产品。包括文档标准，如生成的需求文档的结构；文档编写标准，如定义对象类时注释标题的标准写法；还有编码标准，规定如何使用某种编程语言。
2. 过程标准。这些标准定义了软件开发必须遵循的过程。应将良好的开发实践封装其中。过程标准包括对规格说明的定义、设计、确认过程、过程支持工具以及对在这些过程中产生的文档的描述。

图 24-4 展示了几个可能被使用的产品标准和过程标准的例子。

产品标准	过程标准
设计评审表	设计评审方式
需求文档结构	为系统构建提交新代码
方法头格式	版本发布过程
编程风格	项目计划批准过程
项目计划格式	变更控制过程
变更请求表	测试记录过程

图 24-4 产品标准和过程标准

标准必须以提升产品质量的形式表现价值。有的标准需要花费大量时间和劳动，但是只是带来了细微的质量改进，这种标准是没有必要定义的。必须设计可以应用的产品标准，并以成本－效益的方式检查标准；过程标准应该包含用于检查是否遵循产品标准的过程定义。

公司内使用的软件工程标准通常源自更宽泛的国家标准或国际标准。已经制定的国家标准和国际标准涵盖了软件工程术语、编程语言（如 Java 和 C++）、符号系统（如制图符号）、

软件需求的获取分析和书写规程、质量保证规程以及软件验证和确认过程（IEEE 2003）等许多方面。对于安全关键和信息安全关键的系统，制定了更多的专用标准。

软件工程人员有时会把软件标准视为一种行政命令，认为它们与软件开发的技术活动毫不相干，尤其是在标准中要求填写烦琐的文档和工作记录的时候。尽管他们大都承认贯彻实施通用标准是十分必要的，但工程师总能找出一些理由，力图说明某些标准并不适合他们的具体项目。因此，为了让工程师认可标准的价值，设定这些标准的质量管理人员可以考虑采取以下这些措施。

1. 让软件工程师参与产品标准的选择。如果开发者了解了选择标准的原因，就会自觉执行这些标准。最理想的情况，标准文档不应只是列出需要执行的标准，还应该包括评论性的解释，说明为什么得出这样的标准化的决议。

2. 定期评审和修改标准以反映技术的变化。制定标准代价不菲，标准一经制定出来就要载入公司的标准手册。由于成本和所需要的讨论，通常不会轻易对标准进行改动。标准手册是必备的，但是它要随着环境和技术的变化而不断完善。

3. 尽可能提供工具来支持基于标准的开发。遵循标准常常涉及一些乏味的通常可由软件工具完成的手工工作，因此，开发人员经常觉得标准是大麻烦。如果有工具支持，遵循软件开发标准就只需要额外付出很少的精力。例如程序框图可以通过一个语法制导的程序编辑系统来定义和实现。

不同类型软件需要不同的开发过程，所以必须采用适当标准。如果某种工作方式不适合一个项目或项目团队，对它做出规定是没有意义的。因此每个项目管理者都应该有根据个别情况改动标准的权力。然而，当做出变更时，保证这些变更不会影响产品质量是很重要的。

项目管理者和质量管理者可以通过切实可行的质量规划避免标准的不适当问题。他们应该确定质量手册中哪些标准应该不折不扣地执行，哪些标准应该修改，哪些标准应该废止。对于某些用户和特定的项目需求，可以制定相应的新标准。例如，如果以前的项目中没有用到形式化规格说明的标准，就需要制定这些标准。

24.2.1 ISO 9001 标准框架

ISO 9000 是一个用于在所有行业建立质量管理体系的国际标准集。ISO 9000 可应用的范围很广，从制造业到服务业都有涉及。ISO 9001 在这些标准中是最具普遍性的，它适用于设计、开发和产品（包括软件）维护等组织内的质量过程。ISO 9001 标准最初开发于 1987 年。下文讨论的是 2008 年发布的 ISO 9001 标准，但是 2015 年发布的标准新版本可能会有所不同。

ISO 9001 标准自身并不是一个软件开发的标准，而是开发软件标准的一个框架。它制定出一般的质量原则，描述一般的质量过程，并且编排应该定义的组织标准和规程。这些应当记录在组织质量手册中。

ISO 9001 标准在 2000 年进行了一次重大修订，形成了 9 个核心过程，如图 24-5 所示。如果一个组织要遵循 ISO 9001 标准，那么必须记录其过程是如何与这 9 个核心过程相对应的。也必须定义和维护有关记录以证明所定义的组织过程已经得到了严格执行。公司的质量手册应该描述相关的过程以及过程数据，这些数据必须收集并得到维护。

ISO 9001 标准并没有定义或规定公司应该使用的特定质量过程。要与该标准一致，公司必须定义过程的类型，如图 24-5 所示，并有相应的流程证明质量过程是得到严格遵守的。这就带来了不同产业部门和公司规模之间的灵活性。

针对正在开发的软件类型，可以定义适合的质量标准。小型公司能够拥有灵活的（不需要

太多文档)但同时仍然服从 ISO 9001 标准的简单过程。但是,这种灵活性意味着我们不能对遵循 ISO 9001 标准的不同公司过程之间的相似性和差异性做出假定。一些公司可能拥有很严格的质量过程来保留详细的记录,而另一些公司可能不那么正式,只有极少量的附加文档。

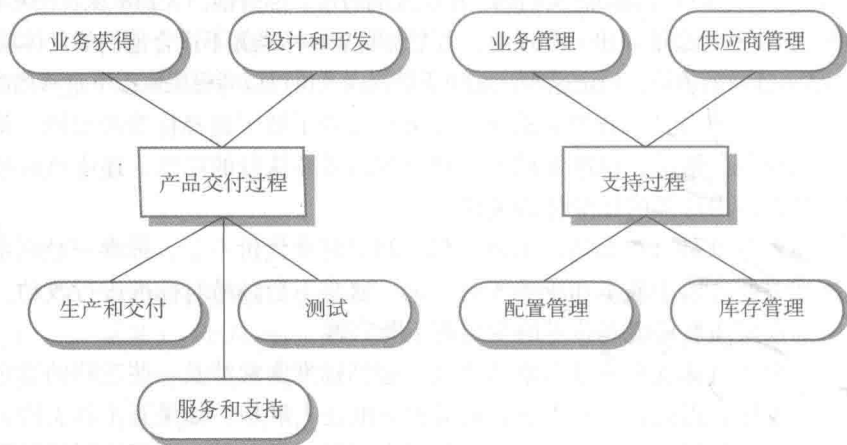


图 24-5 ISO 9001 核心过程

ISO 9001、组织的质量手册和单个项目质量计划之间的关系如图 24-6 所示。这张图源于 Ince(Ince 1994)提出的一个模型,他解释了通用的 ISO 9001 标准如何作为一个软件质量管理过程的基础来使用。Bamford 和 Deilbler(Bamford and Deibler 2003)解释了后来的 ISO 9001:2000 标准如何在软件公司中被采用。

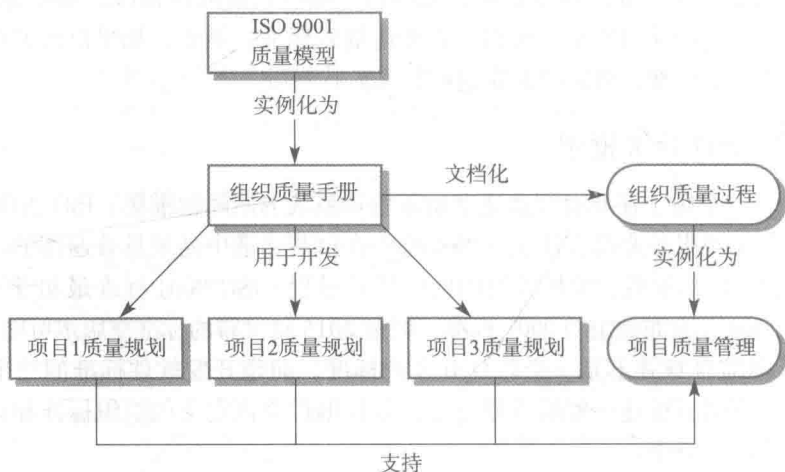


图 24-6 ISO 9001 和质量管理

一些软件客户要求他们的供应商获得 ISO 9001 认证。软件开发公司拥有经过认证的质量管理系统,客户才会有信心。拥有独立鉴定资格的机构对质量管理过程和过程文档进行检查,判断这些过程是否符合 ISO 9001 标准的所有内容。如果的确如此,就像质量手册里定义的一样,他们认证这家公司的质量过程符合 ISO 9001 标准。

有些人认为 ISO 9001 证书意味着经过认证的公司所生产的软件质量比未经认证的公司生产的要好,这是一种误解。ISO 9001 标准只是关心组织里拥有质量管理规程,并且遵循

这些规程，并没有保证通过 ISO 9001 认证的公司使用最好的软件开发实践，或是这些过程会产生高质量的软件。

ISO 9001 认证并不充分，因为它将质量定义为符合标准，但不考虑软件用户的体验质量。比如，某家公司所定义的测试覆盖标准是，对象中所有方法必须使用至少一次。不幸的是，不完整的测试也能满足这个标准，例如对方法的不同参数没有进行测试。只要遵循了定义的测试规程，并且维护了测试记录，这个公司就是满足 ISO 9001 认证的。

24.3 评审与审查

评审 (review) 与审查 (inspection) 是检查项目可交付物质量的 QA 活动。这涉及检查软件、文档以及审查过程的记录，以发现错误和遗漏，还要检查是否遵循质量标准。第 8 章提到，评审、审查和程序测试是软件验证和确认 (V&V) 通用过程的一部分。

在评审过程中，一个团队检查软件及其相关文档，寻找潜在问题和与标准不一致的部分。评审团队就软件质量水平或项目文档做出判断。然后项目管理人员使用这些评定来做出规划决策并为开发过程分配资源。

质量评审基于软件开发中产生的文档来进行。软件规格说明、设计、代码、过程模型、测试计划、配置管理规程、过程标准以及用户指南，这些可能都要评审。评审应当检查文档和代码的一致性和完整性，确保遵循质量标准。

然而，评审不仅仅是检查与标准的一致性，还被用来帮助发现软件和项目文档中的问题和遗漏。评审的结果应当作为质量管理过程的一部分被正式记录。如果发现了问题，应将评审人员的意见交给开发者或者负责修改所发现问题的人员。

评审和审查的目的是提升软件的质量，不是评估开发团队中员工的表现。相对于较为私下进行的构件测试过程，评审是一个检测错误的公开过程。不可避免的是，个人犯的错误会暴露在整个开发团队面前。要确保所有开发人员对评审过程起到有建设性的作用，项目管理人员必须对个人的关注点保持敏感。他们必须营造一种工作文化，即发现错误时不责备当事人。

尽管质量评审为制定管理决策提供关于软件质量的信息，但是质量评审不同于管理过程评审。过程评审将软件项目的实际过程与计划过程对比，主要关注项目是否能够按时并在预算范围内发布有用的软件。过程评审将外部因素考虑在内，环境变化可能导致不再需要开发软件或是必须做出彻底改动。由于业务或它的操作环境发生了改变，导致已开发出高质量软件的项目不得不被撤销。

24.3.1 评审过程

尽管在评审的细节上有很多不同，但是评审过程 (见图 24-7) 一般分为下面 3 个阶段。

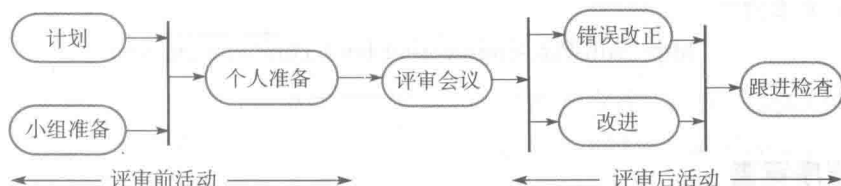


图 24-7 软件评审过程

1. 评审前活动。这些准备工作对于评审的有效进行是必需的。具有代表性的是, 这些评审前工作关心评审的计划和评审的准备工作。评审计划包括建立一个评审团队, 安排评审的时间和地点, 分发要评审的文档。在评审准备工作中, 评审团队会得到要评审的软件的一个综述。评审团队各个成员需要阅读并理解软件、文档以及相关的标准。他们独立地工作, 依靠标准找出错误、遗漏和不符合的地方。评审人员如果不能参加评审会议, 他们可以针对软件提出书面意见。

2. 评审会议。在评审会议期间, 被评审的文档或程序的作者应该和评审团队一起把文档从头到尾浏览一遍。评审本身的时间相对而言不长, 至多 2 小时。一个团队成员应该作为评审的主席, 还应该有一个成员正式记录所有评审决议和要采取的行动。在评审期间, 主席负责保证所有的书面意见都被考虑在内。评审主席应该在评审期间写下一个达成共识的意见和行动的记录。

3. 评审后活动。在评审会议结束后, 必须解决在评审期间提出的问题。这可能包括修复软件漏洞, 重构软件以使它与质量标准相一致, 或是重写文档。有时, 在质量评审中发现的问题要进行一次管理评审, 以决定是否加入更多资源来解决问题。在做出改动之后, 评审主席会检查所有被考虑的评审意见。有时, 要求采取进一步的评审来检查是否所有之前的评审意见都做出了改动。

评审团队应该挑选 3 ~ 4 名主要评审员作为团队的核心, 应该有一个资深设计人员负责做出技术上的重大决策。主要评审员可以邀请其他的项目成员(如相关子系统的设计人员)帮助评审, 他们不必参与整个文档的评审, 而应集中精力解决影响他们工作的问题。另外, 评审团队可以传阅要评审的文档, 并要求其他的项目成员写出书面意见。项目管理人员不需要参与评审, 除非预期的问题要求改变项目计划。

上述的评审过程需要开发团队中所有成员都位于同一地点, 并且可以参加团队面对面会议, 讨论待评审的软件或文档。但是, 现在项目团队通常是分散的, 有时候分布在几个不同国家甚至大洲, 所以团队成员聚到一个屋子里开会通常是不实际的。在这种情形下, 可以使用共享文档来支持远程评审, 每个团队成员使用评论来注释文档。由于时间安排冲突或者团队成员在不同时区工作等原因, 面对面的会议是不可能进行的。评审主席有责任协调评审评论, 以及单独与评审团队成员讨论所做的修改。



审查过程的角色

当 IBM 首次建立程序审查的时候 (Fagan 1986), 对审查小组的成员有多个正式的角色分工。这些角色包括协调员、代码阅读者、抄写员。审查实践中其他用户修改了这些角色, 但是通常都接受审查中要有以下角色: 代码作者、督察员、抄写员、协调员。审查过程由协调员来主持。

<http://software-engineering-book.com/web/qm-roles>

24.3.2 程序审查

程序审查是“同行评审”, 团队成员合作发现已开发程序中的漏洞。第 8 章中谈到, 审

查可以作为软件验证和确认 (V&V) 过程的一部分。因为它们不要求执行程序, 所以它们和测试互补。这就意味着程序审查能够验证系统的不完整版本, 并且能够检查像统一建模语言 (UML) 模型这样的表示法。程序测试可能被评审。测试评审能够发现测试的问题, 并且提升这些测试检测程序错误的有效性。

程序审查涉及来自不同背景的团队成員, 他们对程序源代码进行精心的、逐行的评审。他们寻找错误和问题, 并且在审查会议中陈述出来。缺陷可能是逻辑错误, 也可能是代码中的异常, 这些异常可能表明了错误情况或者代码遗漏的功能特征。评审团队详细检查设计模型和程序代码, 并且标记出需修正的异常和问题。

审查时, 经常使用一份常见编程错误的检查表。这份检查表是基于来自书本的实例和个别应用领域的错误经验做出的。我们对于不同的编程语言使用不同的检查表, 因为每种语言有它自己特有的错误。Humphrey (Humphrey 1989) 在对审查的详细讨论中, 给出了多个检查表的例子。

在审查过程中可能做出的检查如图 24-8 所示。每个组织都应当根据部门标准和实践开发自己的检查表。由于不断发现新的错误类型, 这些检查表应经常更新。因为编译时会有不同的检查级别, 检查表的条目根据编程语言不同而有区别。例如, Java 编译器检查函数的参数个数是否正确, 但是 C 编译器却不检查。

故障分类	审查内容
数据故障	<ul style="list-style-type: none"> • 所有的程序变量都在使用前被初始化了吗? • 所有的常量都命名了吗? • 数组的上边界应该等于数组长度还是长度减 1? • 如果使用字符串, 定位符是显式指定的吗? • 有缓冲区溢出的可能性吗?
控制故障	<ul style="list-style-type: none"> • 对每一个条件语句, 条件是正确的吗? • 每一个循环都能终止吗? • 复合语句被正确地括起来了吗? • 对于 case 语句, 所有可能的情况都考虑到了吗? • 若每一个 case 语句都需要跟一个 break 语句, 有遗漏吗?
输入 / 输出故障	<ul style="list-style-type: none"> • 所有的输入变量都使用了吗? • 所有的输出变量在输出前都被赋值了吗? • 未料到的输入会引起系统崩溃吗?
接口故障	<ul style="list-style-type: none"> • 所有的函数和方法调用的参数数量都正确吗? • 形参和实参类型匹配吗? • 参数顺序都对吗? • 如果构件访问共享内存, 它们都有相同的共享内存结构模型吗?
存储管理故障	<ul style="list-style-type: none"> • 如果一个链接结构被修改了, 所有的链接都得到重新赋值了吗? • 如果使用了动态存储, 空间分配正确吗? • 如果空间不再使用, 显式地释放空间了吗?
异常管理故障	<ul style="list-style-type: none"> • 所有可能的错误状态都已经考虑到了吗?

图 24-8 审查过程中的检查表

大多引入审查的公司发现, 审查在发现漏洞方面是非常有效的。据 Fagan (Fagan 1986) 报告, 使用非形式化的程序审查可以检测到 60% 以上的程序错误。McConnell (McConnell 2004) 对比了单元测试, 其错误探测率大约为 25%; 而通过审查, 错误探测率达到了 60%。这些比较是在自动化测试广泛流行之前完成的, 所以不清楚审查与自动化测试之间的比较。

尽管评审和审查具有良好的成本-效益,很多软件开发公司还是不愿意使用审查或者同行评审。审查会比测试更有效地发现错误,有程序测试经验的软件工程师有时会不愿意接受这一观点。管理人员也可能产生怀疑,因为在设计和开发过程中审查工作要求额外的开销。他们不希望冒风险,即评审和审查没有在程序测试中带来相应的成本节省。

24.4 质量管理与敏捷开发

软件工程中的敏捷方法关注代码的开发。这些方法使与代码开发没有直接关系的文档和过程减到最少,并且强调团队成员之间的非正式沟通而不是基于项目文档的沟通。敏捷开发中的质量意味着代码质量和实践活动,比如使用重构和测试驱动开发确保产出高质量代码。

敏捷开发中的质量管理是非正式的,而不是基于文档的。它依赖于建立一种质量文化,这种文化下所有团队成员都对软件质量抱有责任感,并且会采取措施来维护质量。敏捷社区从根本上反对 ISO 9001 中基于标准的方法和质量过程所带来的“官僚”成本。采用敏捷开发方法的公司很少关注 ISO 9001 认证。

在敏捷开发中,质量管理基于良好实践的共享,而不是正式文档。下面列出几个良好的实践例子。

1. 提交之前进行检查。程序员有责任在提交代码构建系统前,与其他团队成员一起对自己的代码进行代码评审。
2. 决不破坏构建。团队成员提交的代码引发系统整体崩溃是不可接受的。因此,每个人必须针对整个系统对他的代码变更进行测试,并且确信这些代码能够按预期工作。如果构建被破坏了,错误代码的引入者应该优先解决这个问题。
3. 发现问题就修复它。系统代码属于整个团队而不是个人。因此,如果一个程序员发现别人的代码出现问题或者过于晦涩难懂,他可以直接修复这些问题,而不需要让代码的原作者解决。

敏捷软件开发中的评审过程通常是非正式的。例如,在 Scrum 中,在每次软件迭代完成后都会有一个评审会议(冲刺评审),其中会讨论到质量问题。为了避免引起质量问题,团队可以决定改变其工作方式。在冲刺评审期间,集体决策关注重构和质量改进,而不是增加新的系统功能。

代码评审可能是个人的责任(提交之前进行检查),也可能依赖于结对编程的使用。第3章中提到,结对编程是指两个人一起工作,共同进行代码开发并共同对代码负责。这种编程方式下,一个人开发的代码不断被另一个人检验和评审。两个人共同查看每一行代码并在代码被接受前进行测试。

结对编程会使人员对程序有一个深入的了解,因为两个程序员必须理解程序的工作细节才能继续开发。这种了解的深度有时很难在其他审查过程中达到,因此结对编程能找出正式审查过程有时都不能发现的漏洞。然而,所涉及的两个人不能像外部审查小组那样客观,因为他们是在检查自己的工作。潜在的问题包括下面这些。

1. 共同误解。结对的两位成员在理解系统需求时可能会犯同样的错误。他们之间的讨论可能会加深错误。
2. 结对声誉。结对小组可能不愿意查找错误,因为他们不想减慢项目的进度。
3. 工作关系。结对小组发现缺陷的能力可能会由于密切的工作关系而降低,因为他们不愿批评工作伙伴。

对于控制软件规格说明的软件开发公司而言,敏捷方法中采用的非正式质量管理方法对于软件产品开发非常有效。没有必要向外部客户提供质量报告,也不需要与其他质量管理团队整合。然而,当为外部客户开发大型系统时,采用尽量少使用文档的敏捷方法进行质量管理可能是不切实际的,原因如下。

1. 如果客户是大型公司,他可能有自己的质量管理过程,并且可能希望软件开发公司用与这些过程兼容的方式来报告开发进度。因此,开发团队可能必须根据客户的要求制定正式的质量规划和质量报告。

2. 如果开发中涉及多个位于不同地方(分布式)的团队,他们可能来自于不同公司,那么非正式沟通可能是不切实际的。不同公司可能有不同的质量管理方法,所以可能需要就质量管理达成共识并提供正式文档。

3. 对于长生命周期的系统,参与开发的团队将会发生变化,如果没有任何文档,新团队成员将难以理解为什么做这些开发决策。

因此,敏捷方法中进行质量管理的非正式方法可能需要调整,以便引入一些质量文档和过程。这个方法通常与迭代开发过程集成。冲刺或迭代之一应该关注产生必要的软件文档,而不是软件开发。

24.5 软件度量

软件度量(measurement)就是对软件系统的某些属性(如复杂度或可靠性)进行量化。在得到的数据之间以及数据和组织通用标准之间进行比较,就可以得出有关软件质量的结论,或者评估软件过程、工具和方法的有效性。在理想情况下,质量管理依赖于对影响软件质量的属性的度量。你可以客观地评估旨在提高软件质量的过程和工具的变更。

举个例子,假设一个公司计划引入新的软件测试工具。在引入这个工具之前,记录下在给定时间内发现的软件缺陷数目。这是评估工具有效性的基线。使用工具一段时间后,重复这个过程。如果引入该工具后在相同的时间内发现的缺陷数目增多,就可以认为这种工具能给软件确认过程提供有益的支持。

软件度量的长期目标是利用度量来对软件质量进行评判。理想情况下,使用一系列量度对软件属性进行度量能评估一个系统,通过度量可以推断出系统的质量水平。如果一个软件达到了所需的质量阈值,那么它就可以不通过评审而被接受。适当的情况下,度量工具还可以突出显示出软件需要改进的部分。然而,现实距离理想情况还相距甚远。想要达到自动质量评估的理想状况在可预见的将来还不太可能。

软件量度(metric)是能够被客观度量的软件系统、系统文档或开发过程的特性。量度的例子包括:以代码行数表示的软件产品规模;雾(fog)指数,它是一段文本的可读性的指标;交付的软件产品中所报告的故障数;开发一个系统构件所需的人日数等。

软件量度可以是控制量度或是预测性量度。顾名思义,控制量度支持过程管理,而预测性量度帮助预测软件的特性。控制量度通常与软件过程相关。修复已发现缺陷所需平均工作量和时间是控制量度(过程量度)的例子。有3种过程量度可以使用。

1. 完成特定过程所需的时间。这个时间可以是用于该过程的总时间、日历时间、特定工程师花费在该过程上的时间等。

2. 特定过程所需的资源。资源可能包括以人日为单位的总工作量、差旅费或计算机资源。

3. 特定事件的发生次数。可能被监控的事件例子包括：代码审查期间发现的缺陷数量，请求的需求变更的数量，已交付系统中的错误报告的数量，以及为响应需求变更而修改的代码行的平均数。

预测性量度又叫产品量度，与软件本身相关。预测性量度的例子有：模块的环路复杂度，程序中标识符的平均长度，在设计中与对象类有关的属性和操作的数量。无论控制量度还是预测性量度，都能影响管理决策的制定，如图 24-9 所示。管理者使用过程量度来决定是否做出过程改变，使用预测性量度来决定软件变更是否必要以及软件是否可以发布。

本章侧重于讨论预测性量度，通过分析软件系统代码或文档自动评估其价值。网上第 26 章讨论控制量度及其在过程改进中的使用。

软件系统度量可能用到下列两种方法。

- 1. 给系统质量属性赋值。通过度量系统构件的特性，并将这些度量综合起来，就能评估系统质量属性，比如可维护性。
- 2. 识别质量低于标准的系统构件。度量能识别那些特性背离某些规范的个别构件。例如，可以度量构件以发现那些有着最高复杂性的问题构件。由于复杂度高难于理解，这些构件更可能包含错误。

就像图 24-2 所示，直接度量软件的某些质量属性是非常困难的。像可维护性、可理解性和可用性等质量属性是外部属性，与开发者和用户使用软件的经验有关。它们受到主观因素的影响，比如用户的经验和知识使他们无法客观地度量软件。为了对这些属性做出判断，开发者不得不度量软件的某些内在属性（如软件的规模、复杂度等），并假定在所能度量的属性和想要了解的质量特性之间存在着一定的关系。

图 24-10 给出了某些软件的外部质量属性与与其有关的一些内部属性。该图说明在外部属性和内部属性之间会存在某些关系，但没有说明这些属性是如何关联的。内部属性的度量能否对外部的软件特性做出有益的预测，取决于以下 3 个条件（Kitchenham 1990）。

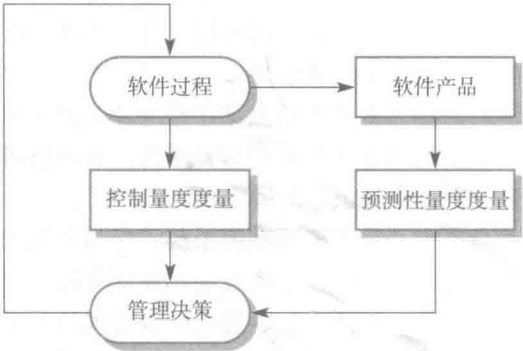


图 24-9 预测性量度和控制量度

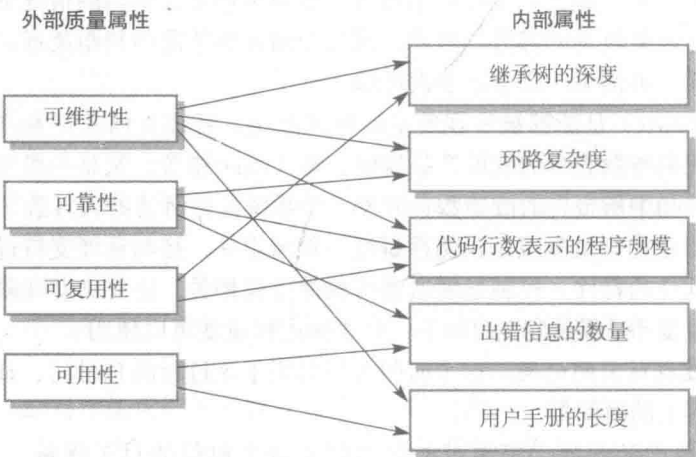


图 24-10 软件内部属性与外部属性的关系

1. 内部属性必须被精确度量。这个往往并不容易,可能需要开发特殊工具来实现相应的度量。

2. 在能够度量的属性和我们感兴趣的外部质量属性之间必须有一定关系。也就是说,在某种程度上,质量属性的值必须和可度量属性的值相关联。

3. 内部属性和外部属性的关系必须是可理解的、可确认的,能用公式或模型表达出来。模型的公式化表示需要识别模型的函数形式(线性的、指数的等),这是通过分析收集到的数据,找出模型中要包含的参数,并用现有数据校正这些参数来完成的。

最近在软件分析领域的研究工作(Zhang et al. 2013)已使用数据挖掘和机器学习技术来分析软件产品和过程数据的存储库。软件分析(Menzies and Zimmermann 2013)背后的思想是,我们实际上不需要一个反映软件质量和已收集数据之间关系的模型。相反,如果有足够的数据,则可以发现相关性并对软件属性做出预测。24.5.4节将讨论软件分析。

目前几乎没有关于工业上系统化软件度量的公开出版信息。许多公司的确收集了软件信息,例如需求变更数或在测试中发现的缺陷数。然而,还不清楚他们是否接下来系统地使用了这些度量去比较软件产品和过程,或是评估变更对软件过程和工具的影响。系统性的度量比较困难的原因如下。

1. 引入一个组织量度或软件分析程序的投资回报是无法量化计算的。在过去的几年里,软件的质量已经取得了极大的改善,这是在并没有使用量度的情况下取得的,因此很难判断引入系统化软件度量和评估的初始开销。

2. 现在还没有软件量度的标准,也没有标准化的度量和分析过程。在这些标准和支持工具出现之前,大多数公司不愿意引入度量。

3. 度量可能需要开发和维护专门的软件工具。当度量的回报未知时,很难证明工具开发的成本是合理的。

4. 在许多公司,软件过程是非标准化的,没有很好定义,而且也很难控制。在某种意义上讲,在同一家公司内部以有效方式使用度量还存在很多过程上的不确定性。

5. 许多关于软件度量和量度的研究主要侧重于基于代码的量度和计划驱动开发过程。然而,越来越多的软件开发是复用和配置现有应用系统或使用敏捷方法,因此,开发者并不知道先前对量度的研究是否适用于这些软件开发技术。

6. 引入度量增加了过程中额外的开销。这和敏捷方法的目标相矛盾,敏捷方法推崇消除那些和程序开发没有直接关联的过程活动。因此已经采取敏捷方法编程的公司并不倾向于采用度量程序。

软件度量和量度是经验软件工程的基础。这是一个新的研究领域,通过对软件工程的实验和对真实项目数据的收集,建立和验证关于软件工程方法和技术的假设。致力于这个领域的研究者们认为,只有当我们能够提供具体的证据来证明某些软件工程方法和技术真正如它们的提出者所声称的那样带来效益,我们才能相信此软件工程方法和技术。

然而,经验软件工程的研究对软件工程实践没有产生重大影响。很难将一般性的研究与不同于研究工作的单个项目联系起来。许多局部因素可能比一般的经验结果更重要。为此,软件分析的研究人员认为,分析不应该试图得出一般性的结论,而应该提供面向特定系统的数据分析。

24.5.1 产品量度

产品量度是用来量化一个软件系统内部属性的预测性量度。产品量度的例子包括系

统规模、代码行数、每个对象类的方法数等。不幸的是，如本节前面讨论过的，容易度量的软件特性（如规模和环路复杂度）与质量属性（如可理解性、可维护性等）之间没有一个清晰而又一致的关系。这种关系是随着开发过程、技术以及被开发系统类型的不同而不同的。

产品量度分为两类。

1. 动态量度，这是通过对执行中的程序进行度量所收集到的。在系统测试期间或系统投入使用后可以收集到这些量度。例如，出错报告的数量或完成计算所花费的时间。

2. 静态量度，这是通过对系统各种表现形式（如设计、程序或文档等）进行度量所收集到的。静态量度的示例如图 24-11 所示。

这些量度类型与不同的质量属性有关。动态量度用于评估一个程序的效率和可靠性，而静态量度则用于评估一个软件系统或系统构件的复杂度、可理解性和可维护性。

动态量度与软件质量特性的关系通常较为清晰。度量特定函数的执行时间和评估系统启动时间相对比较容易，它们与系统的效率有直接的关系。同样，系统失效数和失效的类型要记录下来，它们直接关系到软件的可靠性，如第 12 章讨论的。

正如图 24-11 所示，静态量度与质量属性的关系是间接的。人们已提出很多这类量度，并进行实验，试图得出和验证这些量度与系统的复杂度和可维护性之间的关系。这些实验都没有结论，但根据程序规模和控制复杂度能对可理解性、系统复杂度和可维护性做出最可靠的预测。

软件量度	描 述
扇入 / 扇出	扇入是对调用其他函数或方法的函数数或方法数（假设用 X 表示）的度量。扇出是被 X 调用的函数数。一个高的扇入值意味着 X 与其他的设计紧密结合，对 X 的修改将产生广泛的影响。因为协调被调用构件所需的控制逻辑的复杂度，一个高的扇出值意味着 X 的整体复杂度可能很高
代码长度	这是对程序规模的度量。通常一个构件的代码越多就越复杂，并容易出错。代码长度是预测构件中易出错程度的最可靠的量度之一
环路复杂度	这是对程序控制复杂度的度量。这个控制复杂度可能与程序的可理解性有关。第 8 章介绍了怎样计算环路复杂度
标识符长度	这是对程序中标识符（变量名、类名、方法名等）的平均长度的度量。标识符越长含义可能就越明确，程序也就越可理解
条件嵌套深度	这是对程序中 if 条件嵌套深度的度量。较深的 if 条件嵌套难以理解并可能容易出错
雾指数	这是对文档中字和句子平均长度的度量。雾指数的值越高，文档就越难以理解

图 24-11 静态软件产品量度

图 24-11 中的量度对所有程序都是适用的，但也提出了专门面向对象的量度。图 24-12 概括了 Chidamber 和 Kemerer（Chidamber and Kemerer 1994）套件（有时称为 CK 套件）的 6 个面向对象量度。虽然这些量度是 20 世纪 90 年代初提出来的，但它们仍然是使用最为广泛的面向对象量度。当创建 UML 图时，一些 UML 的设计工具可以自动收集这些量度值。

El-Amam（El-Amam 2001）在对面向对象量度的评论中，讨论了 CK 量度和其他一些面向对象量度，并得出结论，我们还没有足够的证据来弄清楚面向对象量度是如何与外部软件质量关联的。自他 2001 年的分析之后，情况仍没有真正改变。我们仍然不知道如何使用面向对象程序的量度去得出有关软件质量的可靠结论。

面向对象量度	描 述
每个类的加权方法数 (WMC)	这是每个类中的方法数，是对每个方法根据复杂度进行加权后所计算得到的。因此，一个简单方法的复杂度为 1，一个大而复杂的方法的值将大得多。这个度量值越大，对象类就越复杂。复杂的对象似乎更难以被理解。它们在逻辑上未必是耦合的，因而在一棵继承树中不能有效地作为超类进行复用
继承树的深度 (DIT)	这代表在继承树中的具体层数。子类继承超类的属性和操作 (方法)。继承树越深，设计就越复杂。很多对象类必须理解后才能弄清楚树中叶子节点的对象类含义
孩子数 (NOC)	这是度量类的直接子类数。它度量类层次结构的宽度，而 DIT 代表它的深度。NOC 值大意味着更多的复用。它可能意味着要在确认基类上投入更多的工作量，因为依赖于它们的子类数
对象类间耦合度 (CBO)	当一个类中的方法使用在另一个类中定义的方法或实例时，类间就是耦合的。CBO 意味着类是高度依赖的，因此在一个类上的改变会影响程序中的其他类
对类的响应 (RFC)	RFC 是当类的对象接收到消息时可能对此做出响应的方法数的度量。RFC 是与复杂度相关的，RFC 越高，说明类的复杂度越高，因而它就更容易产生错误
方法内聚缺乏度 (LCOM)	LCOM 是通过计算类中各对方法而得出的。LCOM 是两个数的差，一个数是方法间没有共享属性的方法对数，另一个数是方法间有共享属性的方法对数。该度量值受到广泛争议，有很多变种，不清楚它是否真的比其他度量提供了更多有用的信息

图 24-12 CK 面向对象量度套件

24.5.2 软件构件分析

作为软件质量评估过程的一部分，软件度量过程如图 24-13 所示。对系统的每一个构件都使用一系列量度单独分析。不同构件得出的量度值之间可以进行比较，有时还要与以前项目中收集的历史度量数据进行比较。异常的度量（严重偏离正常值）可能表示相关构件的质量存在问题。

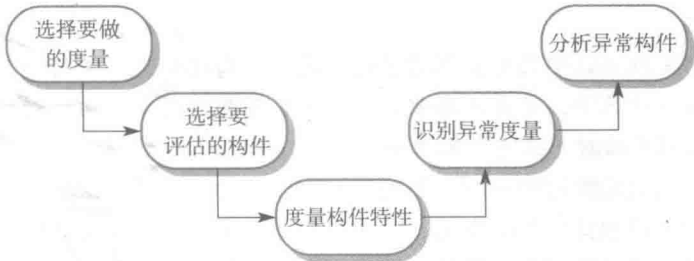


图 24-13 产品度量过程

这个构件度量过程包括下面这些关键阶段。

1. 选择要做的度量。度量要回答的问题应该准确阐述，所需的度量需要良好定义。不必收集与这些问题不直接相关的度量。
2. 选择要评估的构件。在软件系统中评估所有构件的量度值既没有必要，也没有意义。在有些情况下，要选择有代表性的构件进行度量，这样对系统质量有个整体的估计。而在其他情况下，则要评估一些特别关键的构件，例如几乎处于持续使用中的核心构件。这些构件

的质量比那些不经常执行的构件的质量更重要。

3. 度量构件特性。度量选出的构件并计算相应的量度值。这一过程中通常使用一个自动化的数据采集工具对构件的表现形式(设计、代码等)进行处理。这种工具可以是专门写的,也可能是所使用的设计工具的某个特征。

4. 识别异常度量。构件度量一旦完成,就应该对其进行相互比较,还要把它们与度量数据库中的以前的度量相比较。找出每一量度中异常高或者异常低值,从中可以推测出具有这些值的构件可能存在问题。

5. 分析异常构件。一旦从选定的量度中识别出具有异常值的构件,就应该检查这些构件,从而确定这些异常量度值是否意味着该构件的质量出现了问题。复杂度的异常量度值并不一定意味着构件的质量差。特别高的数值可能另有原因,它可能不意味着构件的质量出了问题。

如果可能的话,应该将所有收集的数据作为组织的资源保留,即使在特定的项目中并没有使用这些数据,也应保留所有项目的历史记录。一旦一个非常大的度量数据库建立起来,就可以进行跨项目的软件质量比较,并确认内部构件属性与质量特性的关系。

24.5.3 度量歧义

在收集有关软件 and 软件过程的量化数据时,必须分析这些数据以了解它们的真正含义。曲解了这些数据就很容易得出错误的结论。不能仅仅只了解数据的本身,还必须同时考虑收集数据的上下文环境。

对收集到的数据可能存在不同的解释方式,为了说明这一点,考虑下面的情景,它与系统用户提交的变更请求数有关。

管理者决定度量客户提交的变更请求数。基于这样一种假定,即客户提交的变更请求与产品的可用性和适应性有一定关系,变更请求数越大,软件就越不能满足客户的需要。

处理变更请求和变更软件的费用很高,因此组织决定更改软件过程以提高客户的满意度,同时降低变更的成本。他们希望过程更改会使产品更好,使变更请求减少。进行过程更改,让更多的客户参与到软件设计过程中。对所有产品引入 β 测试,把客户请求的修改反映到交付的产品中。

这个过程做了更改后,继续对变更请求进行度量。由这个更改了的过程所生产的新版本交付给客户。在有些情况下,变更请求数减少;而在有些情况下,反而增加。这使管理者很困惑,不能评估过程更改对产品质量的影响。

为了了解为什么会发生这种事情,必须了解用户为什么会提出变更请求:

1. 软件不能做客户想让它做的事情,因此客户通过请求变更来传达他们所要求的功能;
2. 软件非常好,可以被普遍而又频繁地使用,一些软件客户创造性地想让软件可以完成一些新的功能,因而也可能提出变更请求。

因此,如果客户更多地参与软件开发过程也许会减少客户不满意的地方,从而减少变更请求数量。过程更改比较有效,软件的可用性和适应性会变得更好。然而,过程更改也可能不起作用,客户可能已经决定寻找另一个可供选择的系统。产品因出现竞争对手而失去了市场占有率,也会使变更请求数减少。这样该产品的用户也减少了。

另一方面,过程更改可能使许多新的客户更愿意参与产品的开发过程。因此他们会有更多的变更请求。更改处理变更请求的过程可能会加速变更请求数量的增长。如果公司对客户

积极响应,则用户会产生更多的变更请求,因为他们知道公司认真对待这些要求。这些用户相信他们的建议极有可能会加入到之后的软件版本中。或者,因为 β 测试点选择不当,没有典型地反映出程序的最大多数使用情况,这也会使变更请求数增加。

要分析变更请求数据,我们不能只知道变更请求的数量,还需要知道是谁提出变更请求,他们如何使用该软件,以及为什么会提出该请求。我们还要知道一些外部因素,如更改变更请求规程、市场变化等,是否会对变更请求数量产生影响。有了上述信息,就更有可能会揭示过程更改是否对提高产品质量有意义了。

以上的论述说明,理解变更影响是困难的,解决这个问题的科学方法是减少那些会影响度量的因素。然而,要度量的过程和产品不能孤立于它们的环境而存在,业务环境是不断变化的,而环境的变化可能使数据的对照失去意义。有关人类活动的量化数据不能总是看它的表面值。度量值改变的原因往往是模糊的,应该把度量值之所以能够说明产品质量属性的深层次原因调查清楚。

24.5.4 软件解析

在过去几年中,“大数据分析”概念已经出现,并成为通过自动挖掘和分析大量自动收集的数据来发现规律的一种手段。它可以发现人工数据分析和建模无法找到的数据项之间的关系。软件解析是将这些技术应用于与软件和软件过程有关的数据。

以下两个因素使得软件解析成为可能。

1. 软件产品公司的产品在使用时能够自动收集用户数据。如果软件失效,关于失效和系统状态的信息可以通过因特网从用户的计算机发送到由产品开发者运行的服务器上。因此,可以获得单个产品(如 Internet Explorer 或 Photoshop)的大量数据并用于分析。

2. 开源软件在诸如 Sourceforge 和 GitHub 平台上可以使用,软件工程数据的开源存储库也可以使用(Menzies and Zimmermann 2013)。开源软件的源代码可用于自动分析,并且有时可以链接到开源存储库中的数据。

Menzies 和 Zimmerman (Menzies and Zimmermann 2013) 将软件解析定义为:

软件解析是为管理者和软件工程师提供的软件数据分析,目的是使软件开发人员和团队能够从他们的数据中获取和分享规律,从而做出更好的决策。

Menzies 和 Zimmermann 强调分析的要点不是得出有关软件的一般性理论,而是确定软件开发人员和管理者感兴趣的具体问题。分析旨在实时提供有关这些问题的信息,以便根据分析所提供的信息采取行动。在对微软管理者的研究中,Buse 和 Zimmermann (Buse and Zimmermann 2012) 确定了以下信息需求,例如:如何规定目标测试、审查和重构,何时发布软件,以及如何了解软件客户的需求。

一系列不同的数据挖掘和分析工具可用于软件解析(Witten, Frank, and Hall 2011)。一般来说,对于特定情况不可能知道哪些分析工具最好,必须尝试多个工具来发现哪些是最有效的。Buse 和 Zimmerman 提出了以下工具使用指南。

- 工具应该易于使用,因为管理者可能不太有分析经验。
- 工具应该运行快速并产生简洁的输出,而不是产生大量的信息。
- 工具应该使用尽可能多的参数进行多方面的度量,不可能提前预测出什么规律可能出现。
- 工具应该是交互式的,并允许管理者和开发人员探索分析,它们应该认识到管理者

和开发人员对不同的事情感兴趣。它们不应该是可预测的，而应当基于对过去和当前数据的分析为决策提供支持。

Zhang 和她的同事 (Zhang et al. 2013) 描述了软件解析在性能调试方面的一个极好的实际应用。用户软件可以收集与响应时间和系统状态相关的数据。当响应时间大于预期时，发送该数据用于分析。自动化分析突出了软件中的性能瓶颈。然后，开发团队可以改进算法来消除瓶颈，从而在之后的软件发布版本中提高性能。

在撰写本文时，软件解析还不成熟，现在说它会有什么效果还为时过早。不仅有“大数据”处理的一般问题 (Harford 2013)，而且我们的知识总是依赖于大公司收集的数据。这些数据主要来自软件产品，目前尚不清楚适用于产品的工具和技术是否也可以与定制软件一起使用。小公司不太可能在自动化分析所需的数据采集系统上投资，因此他们可能无法使用软件分析。

要点

- 软件质量管理就是确保软件有较少的缺陷数，并达到可维护性、可靠性、可移植性等既定标准。质量管理活动包括为过程和产品定义标准，并为检测是否符合这些标准而建立过程。
- 软件标准对质量保证来说非常重要，因为这些标准是对“最佳实践”的认同。开发软件时，标准为开发质量优秀的软件提供了坚实的基础。
- 对软件过程产生的可交付物进行评审需要有检查质量标准执行情况的团队人员参加。评审是质量评估的一种最为广泛采用的方法。
- 在程序审查或同行评审中，由一个小团队系统地检查代码。他们仔细地读这些代码并寻找可能的错误和遗漏，然后在代码评审会议中讨论这些问题。
- 敏捷质量管理通常不依赖于独立的质量管理团队。相反，它依赖于建立一种质量文化，使开发团队一起工作来提高软件质量。
- 软件度量可以用于收集有关软件和软件过程的量化数据。所收集的软件度量值可以用于推论产品和过程的质量。
- 产品质量量度对于发现存在质量问题的异常构件具有特别重要的意义。应该对这些构件做更深入的分析。
- 软件解析是对大量软件产品和过程数据的自动分析，从而发现可能为项目管理者 and 开发人员提供规律的关系。

阅读推荐

《Software Quality Assurance: From Theory to Implimentation》是一本优秀的关注软件质量保证理论与实践的著作，包括对如 ISO 9001 之类标准的讨论。(D. Galin, Addison Wesley, 2004)

《Misleading Metrics and Unsound Analyses》是一篇指导量度研究人员理解度量的困难性的好文章。(B. Kitchenham, R. Jeffrey and C. Connaughton, IEEE Software, 24(2), March-April 2007) <http://dx.doi.org/10.1109/MS.2007.49>

《A Practical Guide to Implementing an Agile QA Process on Scrum Projects》这个幻灯片介绍了如何使用 Scrum 将软件质量保证与敏捷开发结合起来。(S. Rayhan, 2008) <https://>

www.scrumalliance.org/system/resource_files/0000/0459/agileqa.pdf

《Software Analytics: So What?》是一篇很好的介绍性文章，解释了什么是软件解析，以及为什么它变得越来越重要。这是一个关于软件解析的特殊问题的介绍，你可能会发现与该问题有关的其他几篇文章来辅助理解软件分析。(T. Menzies and T. Zimmermann, IEEE Software, 30(4), July-August 2013) <http://dx.doi.org/10.1109/MS.2013.86>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap24/>

支持视频的链接: <http://software-engineering-book.com/videos/software-management/>

练习

- 24.1 解释为什么高质量的软件过程会产生高质量的软件产品。讨论这种质量管理体系可能存在的问题。
- 24.2 解释组织如何用标准去获取软件开发的有效方法。提出可在组织标准中获取的 4 种知识。
- 24.3 根据图 24-2 给出的质量属性讨论软件质量评估，依次说明每个属性该如何评估。
- 24.4 简要描述可能用到的标准：
 - 在 C、C# 或 Java 中使用控制结构；
 - 一所大学有关学期项目安排的报告；
 - 程序变更的构建和批准过程（见网上第 26 章）；
 - 购买并安装一个新计算机的过程。
- 24.5 假设你所在的组织要为个人和小型业务开发数据库产品，而该组织对这个软件开发的量化感兴趣，请你写一份报告提出合适的量度，并说明如何收集量度。
- 24.6 为什么程序审查是发现程序错误的一个非常有效的方法？在审查中不可能发现什么类型的错误？
- 24.7 如果在使用敏捷方法开发软件的公司中引入正式的程序审查，可能会出现什么问题？
- 24.8 为什么确认内部产品属性（如环路复杂度）与外部属性（如可维护性）之间的关系是困难的？
- 24.9 假设你在软件产品公司工作，你的经理已阅读关于软件分析的文章，她要求你在这方面做一些研究。调查有关分析的文献，撰写一份简短的报告，总结软件分析中的工作以及引入分析时需要考虑的问题。
- 24.10 有个同事是很棒的程序员，编写的软件错误数很少，但她一贯忽视组织的质量标准。组织的管理者该怎样对待这种行为？

参考文献

- Bamford, R., and W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.
- Buse, R. P. L., and T. Zimmermann. 2012. "Information Needs for Software Development Analytics." In *Int. Conf. on Software Engineering*, 987–996. doi:10.1109/ICSE.2012.6227122.

- Chidamber, S., and C. Kemerer. 1994. "A Metrics Suite for Object-Oriented Design." *IEEE Trans. on Software Eng.* 20 (6): 476–493. doi:10.1109/32.295895.
- El-Amam, K. 2001. "Object-Oriented Metrics: A Review of Theory and Practice." National Research Council of Canada. <http://seg.iit.nrc.ca/English/abstracts/NRC44190.html>.
- Fagan, M. E. 1986. "Advances in Software Inspections." *IEEE Trans. on Software Eng.* SE-12 (7): 744–751. doi:10.1109/TSE.1986.6312976.
- Harford, T. 2013. "Big Data: Are We Making a Big Mistake?" *Financial Times*, March 28. <http://timharford.com/2014/04/big-data-are-we-making-a-big-mistake/>
- Humphrey, W. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.
- IEEE. 2003. *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, CA: IEEE Computer Society Press.
- Ince, D. 1994. *ISO 9001 and Software Quality Assurance*. London: McGraw-Hill.
- Kitchenham, B. 1990. "Software Development Cost Models." In *Software Reliability Handbook*, edited by P. Rook, 487–517. Amsterdam: Elsevier.
- McConnell, S. 2004. *Code Complete: A Practical Handbook of Software Construction, 2nd ed.* Seattle, WA: Microsoft Press.
- Menzies, T., and T. Zimmermann. 2013. "Software Analytics: So What?" *IEEE Software* 30 (4): 31–37. doi:10.1109/MS.2013.86.
- Witten, I. H., E. Frank, and M. A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Burlington, MA: Morgan Kaufmann.
- Zhang, D, S. Han, Y. Dang, J-G. Lou, H. Zhang, and T. Xie. 2013. "Software Analytics in Practice." *IEEE Software* 30 (5): 30–37. doi:10.1109/MS.2013.94.

配置管理

目标

本章的目标是介绍软件配置管理过程和工具。阅读完本章后，你将：

- 了解版本控制系统所应该提供的基本功能，以及这些功能分别在集中式系统与分布式系统上是如何实现的；
- 了解系统构建的挑战，以及持续集成和系统构建的好处；
- 了解软件变更管理的重要性，以及变更管理过程中的基本活动；
- 了解软件发布管理的基础，以及它与版本管理的区别。

在开发和使用过程中软件系统常常会发生变更。开发者必须发现错误并加以修正；系统需求变化后，开发者需要在新版本中实现这些变化。有了新版本的硬件和系统平台之后，开发者需要使自己的系统与之兼容；竞争对手在他们的系统中引入新的特性时，开发者也要做出相应的调整。当软件发生变更时，一个新的版本就产生了。因此，大多数系统都有一系列的版本，每一个版本都需要进行维护和管理。

配置管理（Configuration Management, CM）与管理变更的软件系统的政策、过程和工具有关（Aiello and Sachs 2011）。演化中的系统之所以需要管理，是因为系统在演化时会产生许多不同的版本，这些版本包含了变更提议、故障修正以及对不同硬件和操作系统的适应等诸多内容。可能有几个版本同时开发和使用。这样就需要跟踪已经实现的变更以及这些变更是怎样包含在软件产品中的。如果没有有效的配置管理规程，就可能浪费精力修改一个错误的系统版本，或发布一个错误版本给用户，甚至不知道特定系统或构件源代码存放在什么地方。

配置管理对个人项目管理来说非常有用，因为开发者容易忘记已做过的变更。对于多个开发者同时完成一个软件系统的团队项目来说，配置管理也是必要的。有时这些开发者全都工作在同一个地方，但是，越来越多开发团队的成员分散在世界的各个角落。配置管理系统的使用确保了团队能够访问正在开发的系统，并且能够管理正在执行的变更。

一个软件系统产品的配置管理包括 4 个紧密相关的活动（见图 25-1）。

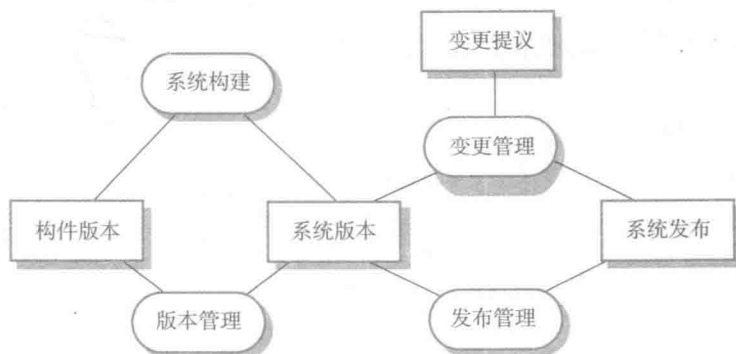


图 25-1 配置管理活动

1. 版本管理。包括跟踪系统构件的多个版本，确保由不同开发者对构件做出的变更不会相互干扰。

2. 系统构建。系统构建是一个组装程序构件、数据和库的过程，这些构件将被编译链接成一个可执行系统。

3. 变更管理。包括跟踪来自客户和开发者的软件变更请求，计算做出这些变更的成本并估计其影响，决定是否变更、何时完成变更。

4. 发布管理。包括准备对外发布的软件，持续跟踪已经发布给客户使用的系统版本。

由于要管理的大量信息以及配置项之间的关系，工具支持对于配置管理而言是必不可少的。配置管理工具用于存储系统构件的版本，从这些构件构建系统，跟踪发布给客户的系统版本，以及跟踪变更提议。这些工具既包括简单工具来支持单一的配置管理任务，例如 bug 追踪，也包括复杂昂贵的集成工具来支持所有的配置管理活动。

敏捷开发中的构件和系统每天要变更多次，如果没有配置管理工具的支持，那么是无法进行敏捷开发的。构件的最终版本保存在共享项目存储库中，开发人员将它们复制到自己的工作空间中。他们对代码进行更改，然后使用系统构建工具在自己的计算机上创建一个新系统进行测试。一旦他们对所做的更改感到满意，就将修改后的构件提交到项目存储库中。这使得修改的构件可供其他团队成员使用。

软件产品或客户软件系统的开发分为以下 3 个阶段。

1. 开发阶段，开发团队负责管理软件配置和加入系统的新功能。开发团队决定系统将要进行的变更。

2. 系统测试阶段，系统的某个版本在内部发布以进行测试。这个阶段由质量管理团队或开发团队中的个人或小组负责。在这个阶段，没有新的功能添加到系统。这个阶段所做的变更是进行缺陷修复、性能改进和信息安全漏洞修复。在这个阶段，可能有一些客户作为 β 测试人员参与进来。

3. 发布阶段，软件发布给客户使用。发布后，客户可以提交缺陷报告和变更请求。可以开发已发布系统的新版本来修复缺陷和漏洞，还可以增加客户建议的新功能。

对于大型系统而言，从来都不仅仅只有一个系统的“工作”版本，在开发的不同阶段会有多个版本的系统。多个团队可能参与不同系统版本的开发。图 25-2 展示了正在开发的系统的如下 3 个版本的情况。

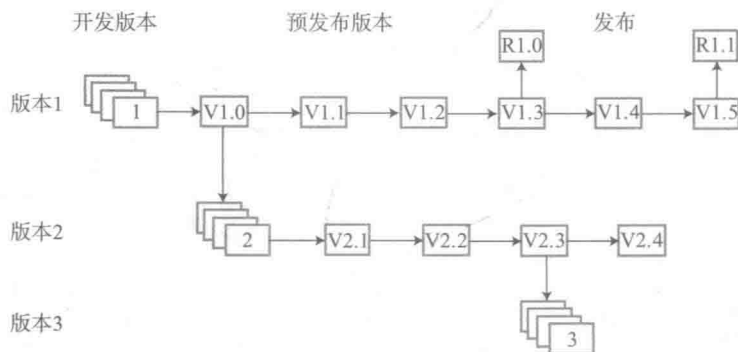


图 25-2 多版本系统开发

1. 版本 1.5 已经开始进行缺陷修复，并且提高第一版发布系统的性能。它是第二个系统

发布版本（R1.1）的基础。

2. 版本 2.4 正在进行测试，以使其成为系统的 2.0 发布版本。在这个阶段没有添加新功能。

3. 版本 3 是一个开发系统，为响应来自于客户和开发团队的变更请求，添加新特征。这将最终作为 3.0 版本发布。

这些不同的版本既有许多通用构件，也有该系统版本所特有的构件或构件版本。配置管理系统跟踪作为每个版本一部分的构件，并在系统构建时根据需要包含它们。

在大型软件系统中，有时配置管理被视为软件质量管理的一部分（第 24 章已经讨论过）。质量管理和配置管理可能是由同一个管理人员负责的。当一个软件的预发布版本完成后，开发人员把软件交给质量管理团队，由他们负责检查系统的质量是否符合要求（是否可接受）。如果符合要求，则该系统变为一个受控的系统，也就意味着在系统接下来所有变更实现之前，都必须经过一致认可并进行记录。

配置管理的难题之一是不同的公司使用不同的术语称呼相同的概念。军用软件系统可能是第一批使用配置管理的系统，这些系统术语反映了在硬件配置管理中使用的过程和术语。商业系统的开发者对军事过程和术语不熟悉，因此发明了他们自己的术语。敏捷方法也引入了新的术语，目的是区别敏捷方法和传统的配置管理方法。图 25-3 给出了一些配置管理术语。

术 语	解 释
基线	基线是组成系统的构件版本的集合。基线是受控的，意味着基线中的构件版本是不能改变的。我们总是可以从它的组成构件中重新创建一个基线
分支	从现存代码线的一个版本创建一个新的代码线。然后新的代码线和已经存在的代码线可以独立开发
代码线	代码线是软件构件以及构件所依赖的其他配置项的集合
配置（版本）控制	确保系统和构件的版本得到记录和维护的过程。这样变更就会得到管理，所有的构件版本就能在整个系统生命期中识别和存储
配置项或软件配置项（SCI）	与配置控制下的软件项目有关的任何事物（设计、代码、测试数据、文档等）。每个配置项都有一个唯一的标识符
主线	代表系统不同版本的基线序列
合并	通过合并不同代码线中的单独版本来创建软件构件的新版本。这些代码线可能是由某个代码线先前存在的分支所创建的
发布版本	发布给客户（或组织内的其他用户）使用的系统版本
存储库	一种共享数据库，存储软件构件的版本信息和与这些构件变更相关的元信息
系统构建	通过耦合和链接构件和库的适当版本来创建一个可执行系统的版本
版本	配置项的一个实例，区别于其他配置项的实例。版本总是有一个唯一的标识符
工作空间	一个私有的工作区域，在其中可以对软件进行修改而不会影响其他使用或修改软件的开发者

图 25-3 配置管理术语

配置管理标准的定义和使用，对于 ISO 9000 和 SEI 的能力成熟度模型（Bamford and Deibler 2003；Chrissis, Konrad, and Shrum 2011）的质量认证至关重要。企业中的 CM 标准可以基于通用标准，例如 IEEE 828-2012，这是一个用于配置管理的 IEEE 标准。这些标准关注 CM 过程和 CM 过程中产生的文档（IEEE 2012）。企业可以在外部标准的基础上经过加工剪裁形成针对其特定需要的更详细、更适合的标准。然而，由于需要大量的文档化开

销，敏捷方法很少使用这些标准。

25.1 版本管理

版本管理 (Version Management, VM) 是持续跟踪软件构件和使用这些构件的系统的不同版本的过程。版本管理也包括确保由不同开发者做出的变更不会相互影响。换句话说，版本管理是管理代码线和基线的过程。

图 25-4 说明了代码线和基线之间的差别。代码线就是源代码的版本序列，一个后期的版本是由某个早期版本演化而来的。代码线通常应用于系统构件，以便每个构件有不同的版本。基线是对一个特定系统的定义。因此基线指的是包含在系统中的构件版本加上所使用的库的描述、配置文件和其他系统信息等。由图 25-4 可以看出，不同的基线使用来自各个代码线的不同构件版本。图中的阴影方框代表了定义在基线中的构件，表明实际上引用了代码线中的构件。主线是由一个原始基线发展而来的系统版本序列。

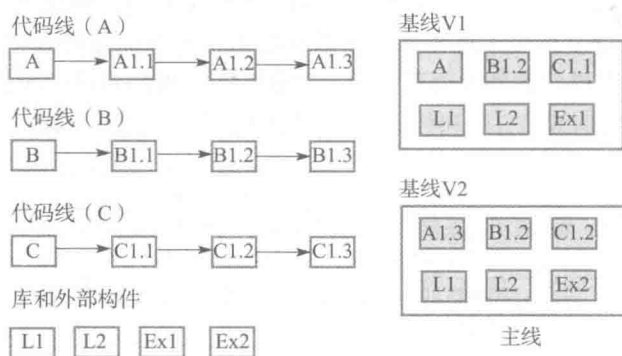


图 25-4 代码线和基线

在实践中可以使用配置语言来刻画基线，以便用户定义一个特定系统版本所包括的构件。精确地描述一个构件版本（比如说 X.1.2）或者仅仅是描述构件标识符（X）都是可行的。若只是简单地在配置描述中使用构件标识符，则意味着在基线中应该使用构件的最近版本。

基线很重要，因为开发者常常不得不重建一个系统的特定版本。例如，一个产品线需要实例化，以便为不同客户产生不同的特定系统版本。假如客户报告了系统中的错误，那么开发者不得不重建这个版本来修复错误。

版本控制 (Version Control, VC) 系统识别、存储和控制对不同版本构件的访问。有两种类型的现代版本控制系统。

1. 集中式系统，其中单个主存储库维护正在开发的软件构件的所有版本。Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) 是一个广泛使用的集中式 VC 系统的例子。

2. 分布式系统，多个用于存储构件版本的存储库同时存在。Git (Loeliger and McCullough 2012) 是一个广泛使用的分布式 VC 系统的例子。

集中式 VC 系统和分布式 VC 系统提供了类似的功能，但以不同的方式实现了这一功能。这些系统的主要特征包括以下这些。

1. 版本和发布版本识别。构件的托管版本在提交给系统时会分配唯一的标识符。这些标识符允许管理同一构件的不同版本，而不需要更改构件名称。还可以为版本分配属性，其中属性集合用于唯一地标识每个版本。

2. 变更历史记录。VC 系统记录并列出了所有对构件做出的变更。在一些系统中, 这些变更可能用来选择一个特定的系统版本。我们可以用描述变更的关键词来标记构件。然后就可以用这些标记来选择包括在基线中的构件。

3. 独立开发。不同的开发者可能同一时间在相同的构件上工作。版本控制系统跟踪检出用于编辑的构件, 确保由不同开发者对构件做出的变更不会相互影响。

4. 项目支持。一个版本控制系统可能支持共享构件的几个项目的开发。可能会需要检入和检出所有与一个项目相关的文件, 而不是一次只能在一个文件或目录上工作。

5. 存储管理。为了减少只有轻微差异的不同版本所占存储空间, 版本控制系统通常使用高效的机制来确保不存储同一文件的多个副本。系统只存储每个版本之间差异的列表而不是每个版本的副本。通过把这些差异应用到源版本(通常是最近的版本)上, 就能够自动重建目标版本。

大多数的软件开发是一个团队活动, 因此经常会有不同的团队成员在同一时间开发同一构件的情况。例如, 假设 Alice 对系统做出一些变更, 其中涉及构件 A、B 和 C。与此同时, Bob 需要变更构件 X、Y 和 C。因此 Alice 和 Bob 都需要变更构件 C。重要的是这些变更不能影响到彼此——Bob 的变更覆盖 Alice 的变更, 或者相反。

为了使彼此之间不相互干涉, 支持独立开发, 版本控制系统使用一个项目存储库和一个私有工作空间的概念。项目存储库维护所有构件的“主”版本, 用于为系统构建创建基线。修改构件时, 开发人员将这些构件从存储库拷贝(检出, check-out)到其工作空间中并处理这些副本。当他们完成其变更时, 将已变更的构件再返回(检入, checked-in)到存储库中。然而, 集中式和分布式 VC 系统以不同的方式支持共享构件的独立开发。

在集中式系统中, 开发人员将构件或构件目录从项目存储库中检出到其私有工作空间, 并在其私有工作空间中对这些副本进行处理。当变更完成后, 他们将构件检入到存储库中。这将创建一个新的构件版本并可以共享。如图 25-5 所示。

在这里, Alice 检出版本 A1.0、B1.0 和 C1.0。她在这些版本上工作, 并创建了新版本 A1.1、B1.1 和 C1.1。她将这些新版本检入到存储库中。Bob 检出 X1.0、Y1.0 和 C1.0。他创建这些构件的新版本, 并将其检入到存储库中。但是, Alice 已经创建了一个 C 的新版本, 而 Bob 一直在使用它。因此, 他的检入创建了另一个版本 C1.2, 以便 Alice 的变更不会被覆盖。

如果两个或更多的人同时在构件上工作, 那么每个人都必须从存储库中检出构件。如果构件已经被检出, 那么版本控制系统会警告其他想要检出该构件的用户这个构件已经被别人检出了。系统还将确保在已修改的构件被检入时, 给不同的版本分配不同的版本标识符并分开存储。

诸如 Git 这样的分布式 VC 系统使用的是另一种方法。在服务器上创建一个“主”存储库来维护开发团队生产的代码。开发人员不是简单地检出他们需要的文件, 而是克隆这个项目存储库, 并在他的计算机上下载和安装。

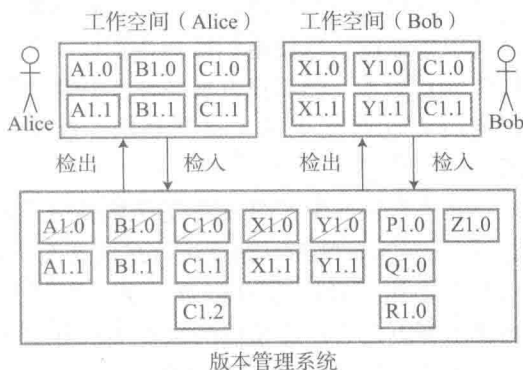


图 25-5 从集中式版本存储库中检入或检出

开发人员在所需的文件上工作，并在他们自己计算机的私有库上维护新版本。完成变更后，他们“提交”这些变更并更新其私有服务器库。然后，他们可以将这些变更“推送”到项目存储库，或告诉集成管理者变更的版本可用。他然后可以将这些文件“拉入”项目存储库（参见图 25-6）。在此示例中，Bob 和 Alice 都克隆了项目存储库并更新了文件。他们还没有把它们推送回项目存储库。

这种开发模式有很多优点：

1. 它为存储库提供了备份机制，就算存储库损坏，也可以继续工作，并且可以从本地副本还原项目存储库；

2. 它允许离线工作，以便开发人员可以在没有网络连接的情况下提交变更；

3. 项目支持是默认的工作方式，开发人员可以在其本地机器上编译和测试整个系统，并测试他们所做的变更。

分布式版本控制对于开源开发至关重要，其中多个人可以在同一个系统上同时工作，而无须任何中央协调。开源系统“管理者”无法知道何时进行变更。在这种情况下，除了维护自己计算机上的私有库，开发人员还维护一个公共服务器库，以便他们推送他们已经变更的构件的新版本。然后由开源系统“管理者”决定何时将这些变更拉入最终系统。此组织如图 25-7 所示。

在这个例子中，Charlie 是开源系统的集成管理者。Alice 和 Bob 独立进行系统开发工作，克隆了最终的项目存储库（1）。除了他们的私有库之外，Alice 和 Bob 都在服务器上维护一个公共库让 Charlie 访问。他们做出变更并测试了变更之后，将变更的版本从私有库推送到他们个人的公共库，并告诉 Charlie 这些库是可用的（2）。Charlie 将这些变更从他们的库中拉入他自己的私有库中进行测试（3）。一旦他对这些变更感到满意，他就会更新最终的项目存储库（4）。

相同构件独立开发的一个后果是可能会出现代码线分支。可能会有几个独立的版本序列，而不是反映变更的一个线性版本序列，如图 25-8 所示。不同的开发者独立地在不同的源代码版本上工作并以不同的方式做出改变，这在系统开发中是很常见的。通常建议在系统上工作时，应创建新的分支，以保证变更不会意外打断正在工作的系统。

在某一阶段，可能需要合并代码线分支以创建一个构件的新版本，使其包括所有已做的变更，这也在图 25-8 中有所体现。图中，构件版本 2.1.2 和 2.3 合并创建了版本 2.4。如果变更发生在代码中不同的位置，版本控制系统会通过综合代码变更使构件版本自动合并。这是添加新特性后的正常操作模式。这些代码变更合并到系统的主副本中。然而，不同开发人员

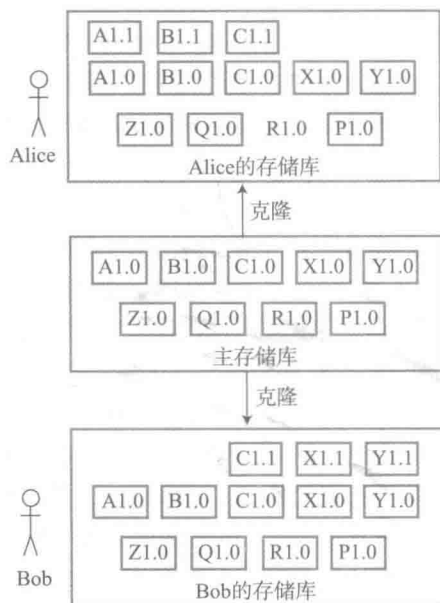


图 25-6 克隆存储库

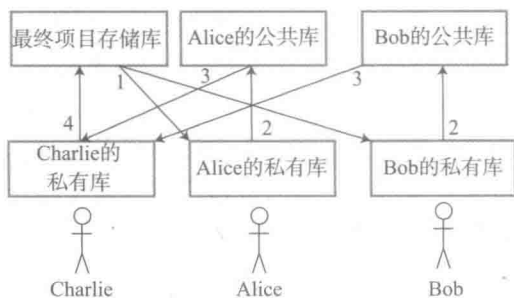


图 25-7 开源开发

所实施的变更有时会重叠。这些变更可能是不相容的，并且会相互干扰。在这种情形下，开发者必须检查出冲突，并且对构件进行修改以解决不同版本间的冲突问题。

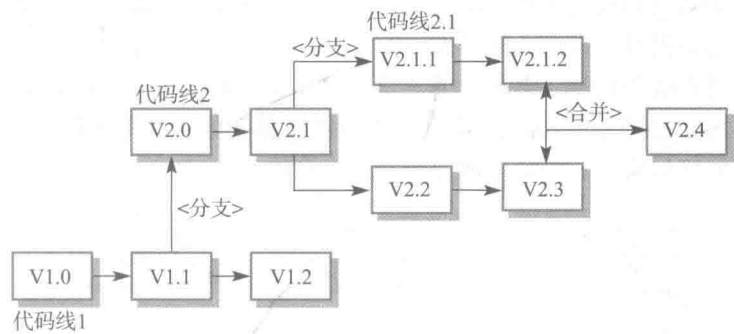


图 25-8 分支和合并

版本控制系统最初推出的时候，存储管理是它们最重要的功能之一。那时候，磁盘空间还很昂贵，因此最小化构件的不同拷贝所使用的磁盘空间十分重要。为此，这些系统不是保存每个版本的完整拷贝，而是保存一个版本之间的差异（增量）列表。通过在一个主版本（通常是最新的版本）基础上应用这些增量，可以重新创建一个目标版本。如图 25-9 所示。

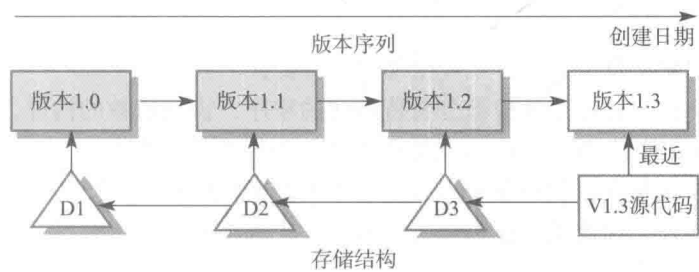


图 25-9 使用增量的储存管理

当一个新版本被创建后，系统简单地存储新版本与用于创建该新版本的老版本之间的一个增量表示（即一个差异列表）。在图 25-9 中，带阴影的方框表示一个构件的老版本，它们是在最新的构件版本基础上自动重建出来的。增量通常被保存为变更行的列表，通过自动应用这些变更信息可以从构件的一个版本创建另一个版本。由于一个构件的最新版本通常都是在使用中的版本，因此大多数系统都完全保存该版本。于是，那些增量就定义了如何重建更早的系统版本。

基于增量的存储管理方法的一个问题是，可能需要很长时间才能应用所有的增量。由于磁盘存储现在相对便宜，Git 使用了一种替代且更快速的方法。Git 不使用增量，而是对存储的文件及其相关的元信息应用标准压缩算法。它不存储重复的文件副本。检索文件时只需要解压缩文件，而无须应用一系列操作。Git 还使用 packfiles 的概念，其中多个较小的文件被组合成一个被索引的单个文件。这减少了与许多小文件相关联的开销。在 packfiles 中使用增量可以进一步减小其规模。

25.2 系统构建

系统构建是通过把软件构件、外部库、配置文件和其他信息编译和链接在一起，创建一

个完整、可执行系统的过程。必须集成系统构建工具和版本控制工具，因为构建过程从版本控制系统管理的存储库中获取构件版本。

系统构建就是将与软件有关的大量信息和它的运行环境装配起来。因此，使用自动构建工具进行系统构建是很有意义的（见图 25-10）。注意，在构建中不只是需要源代码文件，可能还必须将其与外部提供的库、数据文件（比如错误消息的文件）以及定义目标安装的配置文件相链接。可能还必须指明构建所要使用的编译器和其他软件工具的版本。理想情况下，构建一个完整的系统只需要一个命令或者轻点一次鼠标。

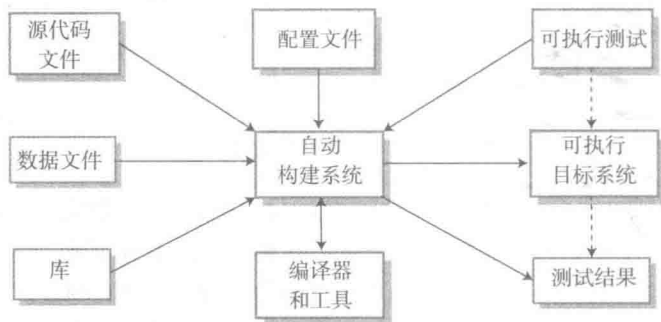


图 25-10 系统构建

系统集成和构建工具包括部分或者全部以下特征：

1. 构建脚本生成。构建系统应该分析待构建的程序，识别依赖的构件，并且自动生成一个构建脚本（配置文件），系统也应该支持手工创建和编辑构建脚本。
2. 版本控制系统集成。构建系统应该从版本控制系统中检出需要的构件版本。
3. 最小化再编译。构建系统应该分析出哪些源代码需要进行再编译，再对需要的代码进行编译。
4. 可执行系统创建。构建系统应该将编译后的各个目标代码文件以及其他需要的文件（比如库文件、配置文件）链接起来，从而创建可执行的系统。
5. 测试自动化。有些构建系统能够使用自动化工具（如 JUnit）运行自动化测试，这样能够检查构建是否被变更所破坏。
6. 报告。构建系统应该提供关于构建或者运行的测试成功与否的报告。
7. 文档生成。构建系统应该能够生成关于构建和系统帮助页面的版本注释。

构建脚本是对要构建的系统的定义，其中包含：构件的信息，构件之间的依赖关系，用于编译和链接系统的工具的版本信息。用来定义构建脚本的配置语言包括一些结构，这些结构描述了构建系统中的构件以及它们之间的依赖关系。

构建是一个复杂的过程，很容易出现错误，可能有如下 3 种不同的系统平台（见图 25-11）。

1. 开发系统，包括开发工具，比如编译器、源码编辑器等。开发人员将代码从版本控制系统下载到私有工作空间，之后再做修改。他们希望在自己的开发环境中建立一个系统版本以便测试，然后再提交做出的修改到版本控制系统。这需要在私有工作空间使用那些本地构建工具去处理检出的构件版本。

2. 构建服务器，用于构建最终的、可执行的系统版本。该服务器维护系统的最终版本。所有系统开发人员检入代码到构建服务器上的版本控制系统进行系统构建。

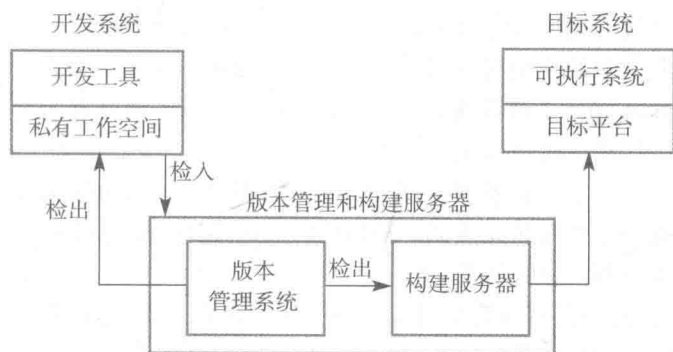


图 25-11 开发、构建和目标平台

3. 目标环境，这是系统运行的平台。这可能是与开发和构建系统所用计算机相同类型的电脑。然而，对于实时和嵌入式系统来说，目标环境相对于开发环境常常更小、更简单（比如手机）。对于大型系统，目标环境可能包含数据库以及其他应用系统，而这些不能安装在开发机器上。在这两种情况下，在开发计算机或者构建服务器上构建和测试系统都是不可能的。

敏捷方法建议使用自动测试执行那些非常频繁的系统构建，从而发现软件问题。频繁构建可以是持续集成过程（见图 25-12）的一部分。为了配合敏捷方法进行大量小修改的理念，持续集成包括在对源代码做出小的修改后，频繁重新构建主线。持续集成的步骤如下。

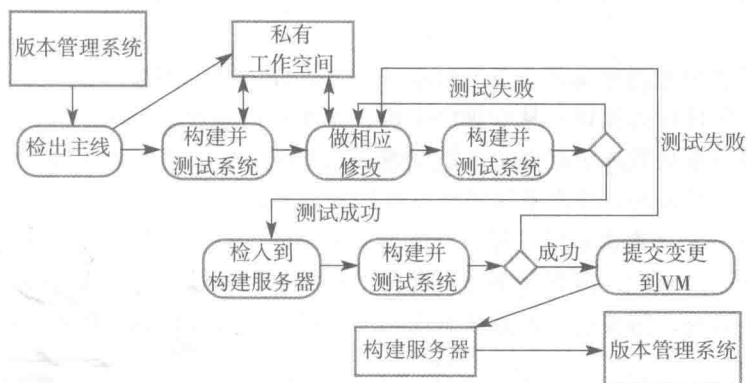


图 25-12 持续集成

1. 将主线系统从版本控制系统中检出到开发人员的私有工作空间。
2. 构建系统并运行自动化测试以确保所构建的系统能够通过所有测试。如果没能通过，构建终止。这时应该通知最后检入基线系统的开发人员，他负责修复这个问题。
3. 完成系统构件的变更。
4. 在私有工作空间构建系统并重新运行测试。如果测试失败，继续编辑。
5. 一旦系统通过测试，将它检入到构建系统服务器，但是不会作为新系统基线检入到版本控制系统。
6. 在构建服务器上构建系统并运行测试。或者，如果使用 Git，可以从服务器中将最近的变更拉入至你的私有工作空间。这样做是因为在检出系统后其他人有可能修改了构件。如

果确实如此, 检出失效的构件并进行编辑, 使测试在私有工作空间通过。

7. 如果系统在构建系统上通过测试, 可将做出的变更作为系统主线中的一个新基线。

像 Jenkins (Smart 2011) 这样的工具能够支持持续集成。当开发人员完成存储库的更新时, 就能安装这些工具从而构建系统。

持续集成的优点是, 它能够发现并且尽快修复由不同开发人员的交互引起的问题。主线中最近的系统是最终的工作系统。然而, 虽然持续集成是一个好想法, 但在系统构建中实现这个方法并不总是可行的, 原因如下。

1. 如果系统很庞大, 构建和测试可能要花费大量时间, 尤其是要与其他应用系统集成时。因此不太可能每天构建系统几次。

2. 如果开发平台和目标平台不相同, 就不大可能在开发人员的私有工作空间中运行系统测试。硬件、操作系统或者安装的软件都可能存在差异。因此需要更多的时间来测试系统。

对于大型系统或者执行平台和开发平台不相同的系统来说, 持续集成可能不太实际。在那些环境下, 可以使用每日构建系统来支持频繁的系统构建。

1. 开发机构设定系统构件的交付时间 (比如下午 2 点)。如果开发人员有正在编写的构件的新版本, 他们必须在下午 2 点前提交。构件可能并不完整, 但是应该提供一些能够测试的基本功能。

2. 通过编译和链接这些构件构建系统的新版本, 形成完整的系统。

3. 将系统交付给测试小组, 执行一系列预定义的系统测试。

4. 记录系统测试中发现的故障, 并交还给系统开发人员。他们在随后的版本中修复这些故障。

频繁进行软件构建的好处是, 更有可能在开发过程早期找到构件交互导致的问题。频繁构建鼓励彻底的构件单元测试。从心理学上讲, 开发人员都处于不要“破坏构建”的压力之下, 即他们都尽量避免检入引起整个系统崩溃的构件版本。因此他们不愿提交未能充分测试的新的构件版本。结果是, 花在系统测试发现和处理软件故障上的时间更少。

由于编译是一个计算密集型过程, 支持系统构建的工具通常被设计成使所需要的编译量最小化。为达到这一目的, 需要检测是否存在可用的已编译版本。如果存在, 就不需要重新编译这个构件。因此, 需要有一种方法来无二义地将构件的源代码和相对应的目标代码联系起来。

解决这个问题的方法是: 赋予存储源代码构件的文件一个独一无二的签名, 对从这个源代码编译的目标文件, 也赋予一个相关的签名。签名能够识别各个源代码版本, 当修改源代码后签名也随之改变。通过比较源代码文件和目标代码文件的签名, 就能够确定是否使用源代码构件产生目标代码构件。

存在两类可以使用的签名, 如图 25-13 所示。

1. 修改时间戳。源代码文件的签名是文件修改的时间和日期。如果构件的源代码文件在相关的目标代码文件之后修改, 系统就认为需要重新编译以生成新的目标代码文件。

比如, 构件 `Comp.java` 和 `Comp.class` 的修改签名分别是 17:03:05:02:14:2014 和 16:58:43:02:14:2014。这表示 Java 代码修改时间是 2014 年 2 月 14 日 17 点 3 分 5 秒, 编译版本的修改时间是 2014 年 2 月 14 日 16 点 58 分 43 秒。这种情况下, 因为编译版本的修改时间早于最新的构件版本, 系统会自动重新编译 `Comp.java`。



图 25-13 链接源代码和目标代码

2. 源代码校验和。源代码文件的签名是从文件中的数据计算出的校验和。校验和函数使用源代码文件作为输入，计算出一个独一无二的值。如果改变源代码（即使是一个字符），就将生成不同的校验和值。这样就能够判断有着不同的校验和的文件一定是不相同的。在编译之前将校验和赋予源代码，从而唯一地标识源文件。然后构建系统将校验和标签附加在生成的目标代码文件上。如果系统中没有与源代码文件签名相同的目标代码存在，就需要重新编译源代码。

由于目标代码文件通常没有版本标识，第一种方法意味着只有最新编译的目标代码文件保存在系统中。目标文件一般通过名字与源代码文件联系起来（比如，和源代码文件有相同的名字，但是加上不同的后缀）。因此，源文件 `Comp.java` 产生目标文件 `Comp.class`。因为源文件和目标文件通过名称联系起来，一般不可能在同一个目录同一时间建立源代码构件的不同版本。由于编译器同时生成目标文件，所以只有最新的编译版本可用。

校验和方法的优点是允许同时保留构件目标代码的多个不同版本。签名（而不是文件名）将源代码和目标代码联系起来。源代码文件和目标代码文件有相同的签名。因此，重新编译构件的时候，不会覆盖目标代码，而时间戳的方式中通常会覆盖原来的代码。编译会生成新的目标代码文件，并贴上源代码的签名。并行编译是可以的，构件的不同版本可以同时进行编译。

25.3 变更管理

变更对大型软件系统而言是一种无法改变的事实。在系统的整个生命周期内组织的需要和需求可能会发生改变，错误需要修正，系统需要适应环境的变化。为了确保变更以一种可控的方式应用在系统中，开发者需要一系列工具的支持和变更管理过程。变更管理的目的是确保系统的演化是一个可控的过程，并且最紧急和成本效益最优的变更拥有最高的优先权。

变更管理是对所提议的变更进行成本效益分析，根据成本效益审批变更请求，并且跟踪系统的哪些构件发生变化的过程。图 25-14 是一个变更管理过程的模型，显示了主要的变更管理活动。当软件交付给客户或是在一个组织中部署时，变更管理过程就开始启动了。

有许多不同的变更管理过程，实际使用哪种取决于软件是一个定制化系统，还是一个产品线，或者是一个成品软件产品。企业的规模也会产生一定的影响，与那些合作客户是其他企业或政府机构的大型企业相比，小型企业会使用不太正式的过程。但是，所有的变更管理过程都应该包含某种形式的变更检查、成本估计和审批。

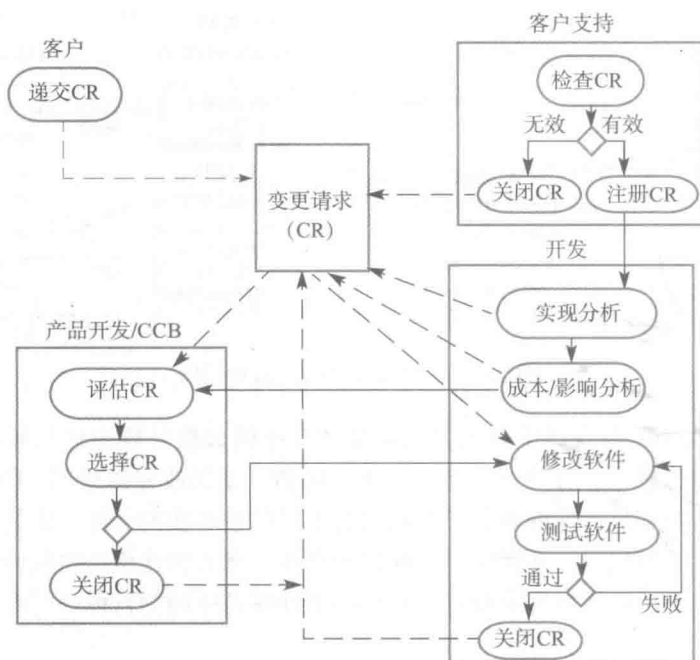


图 25-14 变更管理过程

支持变更管理的工具可能是相对简单的问题或 bug 跟踪系统，或者是集成了用于大规模系统的配置管理包的软件，比如 Rational Clearcase。任何人都可以通过问题跟踪系统报告缺陷或者对系统变更提出建议，并且可以跟踪开发团队是如何回应这些问题的。这些系统不会强加给用户任何一个过程，因此可以在许多不同的设置中使用。围绕着变更管理过程的过程模型，构建了很多复杂的系统。从最初的客户提议到最终变更批准，从变更提交到开发团队处理变更请求，这些复杂系统使上述过程变得自动化。



客户和变更

敏捷方法强调了在变更优先权选择过程中客户参与的重要性。客户代表帮助团队决定在下个开发迭代中需要实现的变更。尽管这对于那些为单个客户开发的系统来说是有效的，但对于那些没有真正的客户与团队一起工作的产品是有问题的。在这样的情况下，团队需要自己决定变更的优先权顺序。

<http://software-engineering-book.com/web/agile-changes/>

当一名利益相关者完成并提交了一个描述对系统的变更的变更请求时，变更管理过程就开始了。这可能是一个描述错误症状的错误报告，也可能是增加系统附加功能的请求。一些企业将 bug 报告和新需求分开处理，但原则上它们都是变更请求。变更请求可以通过填写变更请求表（Change Request Form, CRF）来提交。利益相关者可能是系统所有者和用户、β 测试人员或企业的市场营销部门。

电子变更请求表记录了可供所有参与变更管理的小组分享的信息。当处理变更请求

时，添加到变更请求表中的信息记录了处理过程的每一阶段所做的决定。因此，在任何阶段 CRF 代表变更请求状态的一个快照。除了记录需要的变更之外，CRF 还要记录有关变更的提议、变更的估算成本，以及变更的请求、核准、实现和确认等日期。另外 CRF 还可能包括的一个部分就是开发人员对如何实现变更的概述。再次强调，CRF 的正式程度取决于负责系统开发的组织的规模和类型。

图 25-15 是在一个大型复杂系统工程项目中使用的 CRF 的例子。对于一些更小的项目，变更请求也要正式记录，但是 CRF 可以关注于对所要求的变更的描述，而无须对实现问题进行太多的描述。作为一名系统开发者，必须决定如何实现变更以及估计实现变更所需的时间。

变更请求表

项目: SICSA/AppProcessing

变更请求者: I. Sommerville

请求的变更: 申请者的状态 (拒绝、接受等) 应该以可视化的方式显示在申请人列表中。

变更分析人: R. Looek

受影响的构件: ApplicantListDisplay, StatusUpdater

相关的构件: StudentDatabase

变更评估: 实现比较简单，可以根据状态改变显示颜色。需要增加一个表将状态和颜色关联起来，无须修改任何相关构件。

变更优先级: 中

变更实现:

估计的工作量: 2 小时

到 SGA 应用团队的日期: 28/07/12

决定: 接受变更。变更将在版本 1.2 中实现。

变更实现者:

向 QM 提交的日期:

向 CM 提交的日期:

注释:

编号: 23/02

日期: 20/07/12

分析日期: 25/07/12

CCB 决定日期: 30/07/12

变更日期:

QM 决定:

图 25-15 一个部分完成的变更请求表

变更请求提交后，需要进行检查以确保请求有效。检查者可以来自于客户、应用支持团队，如果是内部请求，检查者可以是开发团队的一个成员。在此阶段，变更请求也可以被驳回。例如，如果变更请求是一个错误报告，这个错误可能已经报告和修复过了。有时，用户认为的问题实际上是由于对系统的误解而提出的。有时，用户请求的一些特性实际上已经在系统中实现了，但他们并不知道。如果上述任一种情况出现，那么变更请求就结束了，并且以结束的原因来更新表格。如果变更请求是有效的，那么就将其记录为一个需要后续分析的未解决请求。

对于有效的变更请求，变更管理过程的下一个阶段就是对变更进行评估和成本估算。这通常是开发团队或者维护团队的任务，因为他们能够计算出完成变更所需的成本。变更对系统其他部分带来的影响必须逐一核查，因此开发者必须检查所有因变更而受到影响的构件。如果变更意味着需要对系统其他部分做进一步的修改，那么这明显会增加完成变更的成本。接下来对所需要的系统模块修改进行评估。最后估算实现变更的成本以及其他系统构件可能要相应发生变更的成本。

由以上分析可知，应该由一个专门的小组来决定软件变更是否是划算的。对于军用和政

府系统，这个小组常被称为变更控制委员会（CCB）。工业上可能被称为“产品开发小组”，他们负责决定一个软件系统如何演化。这个小组应该评审并批准所有的变更请求，除非变更只包括改正屏幕显示或网页和文件中的小错误，否则都应该提交给这个小组，由他们来决定是否应该同意变更。这些“小”的变更请求，应该交给开发小组立即实现。

变更控制委员会或是产品开发小组应从战略和组织的角度，而不应从技术角度去考察变更带来的影响。由他们来决定该变更在经济上是否合理，以及变更的优先级。已通过的变更请求反馈给开发团队，驳回的变更请求就此结束而无须采取进一步行动。在决定是否同意变更请求时，需要考虑的重要因素有以下这些。

1. 不做变更会引起的后果。当评估一个变更请求时，必须考虑如果变更没有完成将会发生什么。如果变更和一个已报告的系统失效有关，则发生失效的严重性必须要考虑到。如果这个系统失效引起系统崩溃，这是非常严重的，不修改可能会影响系统的正常使用。另一方面，如果失效的后果影响较小，比如显示颜色的错误，那么并不需要立即修正这个问题，因此这个变更只拥有较低的优先级。

2. 变更的收益。变更是否使系统的许多用户受益？或者仅仅只令变更的提议者受益？

3. 变更影响的用户数。如果只有一部分用户受到影响，就给变更分配一个较低的优先权。事实上，如果大多数的系统用户都将重新适应所做的修改，那么实现变更可能是不明智的。

4. 变更的成本。如果变更影响许多系统构件（因此增加了引入新错误的机会），并且完成变更需要花费大量的时间，那么该变更可能被驳回。

5. 产品发布周期。如果软件的一个新版本刚刚发布完，那么推迟变更直到计划发布下一个版本，这样做是有意义的。

与针对特定客户开发的软件系统的变更管理相比，针对软件产品（例如 CAD 系统产品）的变更管理在处理上稍有不同。对软件产品，客户并不直接参与有关系统演化的决策，所以变更与客户业务关系不大。这些产品中的变更请求来自客户支持小组、企业市场营销团队和开发团队自身。这些需求可能反映了来自客户的建议和反馈，或者是针对竞争产品所提供的特性的分析。

客户支持小组提交的变更请求可能与软件发布后由客户发现并报告的错误相关。客户可能通过网页或是电子邮件来报告缺陷。缺陷管理小组确认缺陷并把它们写成正式的系统变更请求。市场工作人员会见客户并调查竞争产品，他们建议的变更可能包括如何更容易地向新客户及现有客户推销系统的新版本。系统开发者可能会有一些可以添加到系统中的新特征的好想法。

图 25-14 显示了系统向客户发布后的变更请求过程。开发期间，当系统的新版本通过每日（或更频繁的）系统构建创建以后，就可以采用不那么正式的变更管理过程了。出现的问题和请求的变更仍然要记录在问题跟踪系统中，并在每日的例会上进行讨论。但是只影响到单个构件的变更，会直接送到系统开发人员手中。他们或者接受它们，或者论证为什么这些变更是不必要的。但是，当变更影响到由不同开发团队开发的系统模块时，还是应该由独立的权威人士（例如系统架构师）来评估变更并决定这些变更的优先权。

在一些敏捷方法中，客户直接参与决定是否实现变更。当他们对系统需求提出变更时，他们与项目组成员一起评估变更产生的影响，然后决定这个变更是否应该优先于为系统下一个增量所规划的特征。然而，涉及系统改善的变更就留给系统编程人员来决定了。重构是不断改进软件的过程，不能将其看成是一项额外开销，而应该视为开发过程的一个必要

部分。

当开发团队变更软件构件时，他们应该维护每个构件的变更记录，有时将此称为构件的派生历史（derivation history）。保存派生历史的最佳方式是将其放在构件源代码开头的标准化注释部分（见图 25-16）。这些注释应该引用触发软件系统变更的请求。扫描所有构件派生历史的脚本处理这些注释，之后产生构件变更报告。对于文档，每个版本中的变更记录经常放在一个独立页中，一般放在文档的前面部分（见网上第 30 章）。

SICSA project (XEP 6087)				
//				
// APP-SYSTEM/AUTH/RBAC/USER_ROLE				
//				
// Object: currentRole				
// Author: R. Loock				
// Creation date: 13/11/2012				
//				
// © St Andrews University 2012				
//				
// Modification history				
// Version	Modifier	Date	Change	Reason
// 1.0	J. Jones	11/11/2009	Add header	Submitted to CM
// 1.1	R. Loock	13/11/2009	New field	Change req. R07/02

图 25-16 派生历史

25.4 发布版本管理

系统的发布版本是分发给客户的系统版本。对于大众市场软件，通常定义两种类型的发布版本：一种是主要发布版本，用于交付重要的新功能；另一种是小型发布版本，用于修补漏洞和修复用户报告的问题。比如，本书是在苹果 Mac 电脑上书写的，操作系统是 OS 10.9.2。这表示 OS 10 的第 9 次主要发布的第 2 次小型发布。从经济上来说，主要发布版本对于软件厂商非常重要，因为客户必须对此付款。小型发布版本通常是免费分发的。

系统的发布版本不仅仅是这个系统的可执行代码，还包括：

1. 配置文件，定义对于特定发布版本应该如何配置；
2. 数据文件，比如用不同语言书写的错误信息文件，是成功进行系统操作所必需的；
3. 安装程序，用来帮助在目标硬件上安装系统；
4. 电子和书面文档，用于系统说明；
5. 包装和相关的宣传，为发布版本所做的工作。

对于大众市场产品，准备并分发系统版本是一个花费很高的过程。在准备发布版本的时候，除了必须准备的技术工作之外，还有广告、宣传材料以及到位的市场策略，说服消费者购买新的系统版本。必须认真考虑发布的时间间隔。如果发布过于频繁或者要求硬件升级，客户就可能不愿意升级到新版本，特别是当他们需要付费的时候；如果发布间隔时间较长，会失掉市场份额，因为客户会购买其他的系统。

决定何时发布系统的一个新版本，应该通盘考虑技术和组织的各种具体情况，如图 25-17 所示。

因 素	描 述
竞争	对于大众市场软件，因竞争产品发布新特性而必须发布新的系统版本，因为如果新产品不能提供给已有的客户的话，厂商就可能面临市场份额减少的后果
市场需求	公司的市场部门可能会承诺新的版本在某个特定日期之后可以使用。由于市场的原因，可能必须在新系统中包含新特性，这样才能说服客户去更新他们的版本
平台变更	当新的操作系统版本发布的时候，很可能需要因此而创建软件应用的新版本
系统的技术质量	如果报告的严重系统故障影响到了用户使用系统的方式，发布新版本来修正错误就很可能是必要的。一些细小的系统故障可以通过修补分布在互联网上的代码片段来解决，这种情况就可以在当前的版本上进行

图 25-17 系统发布版本计划的影响因素

发布版本的创建是创建包含系统发布版本所有构件的文件和文档集合的过程。这个过程包含几个阶段：

- 1. 程序的可执行代码及所有相关数据文件都必须在版本控制系统中能够被识别，并标记发布标识符；
- 2. 为不同的硬件和操作系统编写配置描述；
- 3. 为需要配置自己的系统的客户编写更新指令；
- 4. 为安装程序编写脚本；
- 5. 创建网页来描述发布版本，并链接至系统文档；
- 6. 最后，当所有的信息都可用，必须准备该软件的可执行的主映像并移交给客户或销售网点。

对于定制软件或者软件产品线，系统发布管理过程的复杂度与系统用户的数量相关。系统的特定发布版本可能会针对不同的客户，某个客户可以在不同硬件上同时运行几种不同的系统版本。当软件是一个复杂系统之系统中的一部分时，可能需要创建各个系统的若干不同变体。例如，在专门的消防车辆中，每种类型车辆都应该配备适合其设备的软件系统版本。

一个软件企业可能需要管理软件的数十种甚至数百种不同的发布版本。他们的配置管理系统和过程的设计必须能够提供如下信息：哪位客户使用何种系统发布版本，以及发布版本和系统版本的关系。如果交付的系统出现问题，必须能够恢复该特定系统中使用的所有构件版本。

因此，一个系统的发布版本生成时，必须编制文档以保证将来可以准确地重新构建它。这一点对定制的、生命周期长的嵌入式系统尤为重要，比如军事系统或控制复杂机械的系统。这些系统可能有长达 30 年的生命周期。客户可能很多年都使用一个这样的系统，在距离最初发布很长时间之后才对某一特定的软件发布版本提出具体的变更需求。

为发布版本编制文档，必须记录用来产生可执行代码的源代码构件的特定版本。也要保存源代码文件、相应可执行代码以及所有的数据和配置文件的拷贝。幸运的是，这并不意味着旧的硬件总是需要维护。旧的操作系统可以运行在虚拟机上。

还应该记录用来构建这个软件的操作系统的版本、库的版本、编译器的版本和其他工具的版本。这些工具在日后构建完全相同的系统时用得着。因此，平台软件和用来创建系统的工具以及目标系统的源代码都要在版本控制系统中留有备份。

在计划新系统发布版本安装的时候，不能想当然地认为客户总是会安装新的系统版本。一些系统用户可能满足于已有的系统版本，也许他们认为新版本不值得为之付费。因此系统的新版本不能依靠以前的版本的安装。为阐明该问题我们分析下面的情景：

- 1. 系统的发布版本 1 已发布并投入使用；
- 2. 发布版本 2 需要安装新的数据文件，而有些客户不需要发布版本 2 所提供的功能，因

此仍然保留发布版本 1；

3. 发布版本 3 需要发布版本 2 中已安装的数据文件，没有自己新的数据文件。

软件发布者不能假定发布版本 3 所需要的文件都已经在所有站点上安装了。一些站点可能会跳过版本 2，直接从版本 1 升级到版本 3。一些站点可能已经修改了与版本 2 相关的数据文件以反映本地环境的特点。因此，数据文件必须随着系统的版本 3 一起分发和安装。

提供软件即服务 (SaaS) 的一个好处是它避免了所有这些问题。它不仅简化了发布版本管理，还简化了为用户进行系统安装。软件开发人员负责用新版本替换当前的系统版本，这样所有客户就能同时得到新版本。然而，这种方法要求运行服务的所有服务器同时更新。为了支持服务器更新，专门的发布管理工具如 Puppet (Loop 2011) 已经被开发出来，并用于将新软件“推送”到服务器。

要点

- 配置管理是对不断演化的软件系统所做的管理。在维护一个系统时，配置管理团队的作用是保证系统的变更是在受控状态下完成的，并且详细记录已经实现的变更。
- 主要配置管理过程包括变更管理、版本控制、系统构建、变更管理以及发布版本管理。存在能够支持所有这些过程的软件工具。
- 版本控制包括跟踪软件构件由于变更而产生的不同版本。
- 系统构建是将系统构件装配成一个在目标计算机系统中运行的可执行程序的过程。
- 软件应该频繁地重新构建，在新版本构建之后立即进行测试。这样可以更容易发现最后一次构建引入的错误和问题。
- 变更控制包括评估来自系统客户和利益相关者的变更提议，并判断在系统新版本中实现这些变更是否符合成本效益。
- 系统的发布版本包括可执行代码、数据文件、配置文件和文档。发布版本管理包括决定何时发布一个系统，准备好所有待发布的信息，并为每一个系统发布版本编制好文档。

阅读推荐

《Software Configuration Management Patterns: Effective Teamwork, Practical Integration》是一本相对较短且容易阅读的书，在配置管理实践方面尤其针对敏捷开发方法给了很好的具体建议。(S. P. Berczuk 和 B. Appleton, Addison-Wesley, 2003)

《Agile Configuration Management for Large Organizations》这篇网络文章描述可以在敏捷开发过程中使用的配置管理实践，特别强调了如何将这些实践扩展到大项目和大公司。(P. Schuh, 2007) <http://www.ibm.com/developeworks/rational/library/maro7/schuh/index.html>

《Configuration Management Best Practices》是一本很不错的书，提供了比本书讨论的配置管理更广泛的观点，包括硬件配置管理。它面向大型系统项目，并不真正涉及敏捷开发问题。(Bob Aiello and Leslie Sachs, Addison-Wesley, 2011)

《A Behind the Scenes Look at Facebook Release Engineering》是一篇有趣的文章，涉及在云端发布大型系统的新版本问题，在本书中没有讨论这些内容。这里的挑战是确保所有服务器同时更新，以便用户使用的是系统的统一版本而不是系统的不同版本。(P. Ryan, arstechnica.com, 2012) <http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/>

· 《Git SVN Comparison.》这篇文章比较了 Git 和 Subversion 版本控制系统。(2013) <https://git.wiki.kernel.org/index.php/GitSvnComparsion>

网站

本章的 PPT: <http://software-engineering-book.com/slides/chap25/>

支持视频的链接: <http://software-engineering-book.com/videos/software-management/>

练习

- 25.1 如果一个公司没有制定有效的配置管理政策和过程,列举 5 种可能出现的问题。
- 25.2 解释构件的每个版本都必须唯一标识的原因。评述使用简单的基于版本号的版本标识模式的问题。
- 25.3 设想如果两个开发人员同时修改 3 个不同构件,当他们想要合并他们所做出的修改时,可能出现什么问题?
- 25.4 开发软件的团队现在越来越多地在不同地点甚至不同时区工作。列举一些能够支持分布式软件开发的版本控制系统的特征。
- 25.5 列举在系统构建中可能碰到的问题。当在主机上为目标机器构建系统时,可能会发生哪些特殊问题?
- 25.6 关于系统构建,为什么有时有必要维护已经过时的计算机?假设曾经在该计算机上开发过大型软件系统。
- 25.7 在系统构建中一个常见的问题是,将物理文件名包括在系统代码中,而文件名指出的文件结构与目标机器上的文件结构不一致。试写出一组程序设计者指南以帮助他们避免这个问题,列出你能够想到的其他系统构建中容易发生的问题。
- 25.8 在变更管理过程中使用变更请求表作为核心文件的好处是什么?
- 25.9 列举支持变更管理过程的工具应该包含的 6 种特征。
- 25.10 在构建一个大型软件系统的发布版本的过程中,工程师必须考虑哪 5 个因素?

参考文献

Aiello, B., and L. Sachs. 2011. *Configuration Management Best Practices*. Boston: Addison-Wesley.

Bamford, R., and W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.

Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed. Boston: Addison-Wesley.

IEEE. 2012. "IEEE Standard for Configuration Management in Systems and Software Engineering" (IEEE Std 828-2012)." doi:10.1109/IEEESTD.2012.6170935.

Loeliger, J., and M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly and Associates.

Loope, J. 2011. *Managing Infrastructure with Puppet*. Sebastopol, CA: O'Reilly and Associates.

Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly and Associates.

Smart, J. F. 2011. *Jenkins: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

术 语 表

abstract data type (抽象数据类型)。一种由操作而不是表示定义的类型。其表示是私有的, 只能由所定义的操作访问。

acceptance testing (验收测试)。系统的客户对系统进行测试, 以确定其是否足以满足他们的需要从而接受供应商的交付。

activity chart (活动图)。项目管理者用来显示必须完成的任务之间的依赖关系的图。图中显示了任务、完成这些任务的预期时间以及任务间的依赖关系。关键路径是在活动图中最长的路径 (完成任务需要的时间)。关键路径定义了完成项目需要的最短时间。有时也被称作 PERT 图。

Ada (Ada 语言)。一种编程语言, 是 20 世纪 80 年代作为一种开发军用软件的标准语言为美国国防部开发的。它是在 20 世纪 70 年代的编程语言研究基础上开发的, 其中包括诸如抽象数据类型以及并发支持这样的语言元素。它仍然被用于大型、复杂军事和航天系统。

agile manifesto (敏捷宣言)。概括了敏捷软件开发方法背后的基本思想的一组原则。

agile methods (敏捷方法)。适合于快速软件交付的软件开发方法。软件按照增量进行开发和交付, 过程文档和管理开销被最小化。开发关注代码本身, 而不是那些支持文档。

algorithmic cost modelling (算法性成本建模)。一种软件成本估算的方法, 使用一个公式来估算项目成本。公式中的参数是项目和软件自身的属性。

application family (应用族)。具有共同的体系结构和通用的功能的一组软件应用程序。这些应用可以通过修改构件和程序参数来适应特定客户的要求。

application framework (应用框架)。实现一个领域 (例如用户界面) 中的众多应用的公共特征的一组可复用的具体类和抽象类。应用框架中的类在创建应用时进行特化和实例化。

Application Program Interface (API, 应用编程接口)。一种接口, 通常被指定为一组可以访问一个应用程序功能的操作。这意味着其他程序能够直接调用此功能, 而不用通过用户界面访问这些功能。

architectural pattern (style) (体系结构模式 (风格))。已经在很多不同的软件系统中尝试和验证过的一种软件体系结构的抽象描述。模式中所描述的信息包括该模式适合在什么地方使用以及体系结构中构件的组织结构。

architectural view (体系结构视图)。从一个特定角度对软件体系结构的一种描述。

availability (可用性)。当被请求时一个系统提供服务的就绪程度。可用性通常以小数来表示, 因此 0.999 的可用性意味着系统在 1000 个单位时间中有 999 个可以提供服务。

B (B 方法)。一种软件开发的形式化方法, 其基本思想是通过形式化的系统规格说明的系统化转换来实现系统。

bar chart (Gantt chart) (条状图 (甘特图))。项目管理者用来显示项目任务、与这些任务相关的进度, 以及将要参与这些任务的人的一种图。图中显示了任务开始和结束的日期以及按照时间线的人员分配。

black box testing (黑盒测试)。一种测试方法, 其中测试人员无法获得系统或其构件的源代码。测试是根据系统的规格说明来设计的。

BPMN (Business Process Modeling Notation, 业务过程建模符号)。定义描述业务过程和服务

务组合的工作流的符号系统。

brownfield software development (棕地软件开发)。面向存在多个现有系统的环境的软件开发, 所开发的软件必须与这些现有系统集成在一起。

C (C 语言)。一种编程语言, 最初是开发用来实现 Unix 系统的。C 语言是一种相对底层的系统实现语言, 允许访问系统硬件, 并且可以被编译为高效代码。C 语言被广泛应用于底层系统编程以及嵌入式系统开发。

C++ (C++ 语言)。一种面向对象的编程语言, 是 C 语言的超集。

C# (C# 语言)。一种由微软公司开发的面向对象编程语言, 与 C++ 有很多共同之处, 但是包含了允许更多的编译时类型检查的特性。

Capability Maturity Model (CMM, 能力成熟度模型)。软件工程研究所 (The Software Engineering Institute) 的能力成熟度模型, 用于评估一个机构的软件开发成熟度水平。目前已经被 CMMI 所代替, 但仍然被广泛使用。

Computer-Aided Software Engineering (CASE, 计算机辅助软件工程)。20 世纪 80 年代提出的一种概念, 是指使用自动化工具支持开发软件的过程。事实上所有的软件开发现在都依赖于工具支持, 因此这个概念不再被广泛使用。

CASE tool (CASE 工具)。一种软件工具, 例如设计编辑器或程序调试器, 用来支持软件开发过程中的活动。

CASE workbench (CASE 工作台)。一组集成的 CASE 工具, 一起工作来支持主要的过程活动, 例如软件设计或配置管理。现在经常被称为编程环境。

change management (变更管理)。对于所提出的软件系统变更进行记录、检查、分析、估算和实现的过程。

class diagram (类图)。一种描述系统中的对象类以及它们之间关系的 UML 图。

client-server architecture (客户-服务器体系结构)。一种分布式系统的体系结构模型, 其中的系统功能通过服务器提供的一组服务来实现。使用服务的客户端计算机可以访问这些服务。该方法的一些变体 (例如三层客

户-服务器体系结构) 使用多个服务器。

cloud computing (云计算)。在互联网上使用来自外部提供者的服务器“云”提供计算和应用服务。“云”的实现使用了大量的商用计算机和虚拟化技术, 以便有效地使用这些系统。

CMMI。一种集成的过程能力成熟度建模方法, 建立在采用良好的软件工程实践以及集成的质量管理基础上。它支持离散的和连续的能力成熟度建模, 并且集成了系统和软件工程过程成熟度模型。CMMI 是在最初的能力成熟度模型 (CMM) 基础上开发的。

COCOMO II。见 Constructive Cost Modeling。

code of ethics and professional practice (道德规范和职业准则)。定义软件工程师应当具备的道德规范和职业行为的一系列指导方针。由主要的美国专业协会 (ACM 和 IEEE) 制定, 涵盖了 8 个方面的道德规范行为, 包括: 公众、客户和雇主、产品、评价、管理、同事、职业和自身。

Common Request Broker Architecture (CORBA, 通用对象请求代理体系结构)。由对象管理组织 (OMG) 提出的一组标准, 其中定义了一种分布式的构件模型和通信。在分布式系统开发中有影响力, 但现在已很少使用。

component (构件)。软件中可部署的独立的单元, 完全由一组接口来定义和访问。

component model (构件模型)。关于构件实现、文档化和部署的一组标准。其中包括可能由构件提供的特定接口、构件命名、构件互操作以及构件的组合。构件模型为支持构件执行的中间件提供了基础。

Component-based Software Engineering (CBSE, 基于构件的软件工程)。通过独立、可部署、符合构件模型的软件构件的组装来开发软件的方法。

conceptual design (概念设计)。一个复杂系统的高层愿景以及基本能力描述的开发。概念设计能够被非系统工程师的人所理解。

configurable application system (可配置的应用系统)。由系统供应商所开发的一种应用系统产品, 所提供的功能可以通过定制被用于不同的客户企业及环境。

configuration item (配置项)。计算机可读的单

元, 例如一个文档或一个源代码文件, 可以被修改并且所做的修改必须受到配置管理系统的控制。

configuration management (配置管理)。管理对演化中的软件产品的变更的过程。配置管理包括版本管理、系统构建、变更管理和发布版本管理。

Constructive Cost Modelling (COCOMO, 构造性成本建模)。一系列算法性的成本估算模型。最早在 20 世纪 80 年代早期提出, 此后几经修改和更新以反映新的技术以及软件工程实践的变化。COCOMO II 是 COCOMO 最新的版本, 是一种可以免费获得的、有开源软件工具支持的算法性成本估算模型。

CORBA。见 Common Request Broker Architecture。

control metric (控制量度)。一种软件量度, 允许管理人员基于软件过程或者正在开发的软件产品的信息做出规划决策。绝大多数控制量度是过程量度。

critical system (关键性系统)。一旦失效将会造成经济、人员或环境方面的重大损失的计算机系统。

COTS system (商用成品系统)。商用成品系统。商用成品一词现在通常都用于军用系统中。见 configurable application system。

CVS。一种广泛使用的开源软件版本管理工具。

data processing system (数据处理系统)。一种用来处理大量的结构化数据的系统。这些系统通常按照输入 - 处理 - 输出的模式批量处理数据。账务和发票系统以及付款系统都是数据处理系统的实例。

denial of service attack (拒绝服务攻击)。对基于 Web 的软件系统的一种攻击, 试图使系统过载从而使之无法向用户提供正常的服务。

dependability (可依赖性)。系统的可依赖性是一个综合考虑系统的安全性、可靠性、可用性、信息安全性、韧性和其他特性的总体属性。一个系统的可依赖性反映了用户可以信赖该系统的程度。

dependability requirement (可依赖性需求)。一种用于达到所要求的系统可依赖性的系统需求。非功能性的可依赖性需求指定了可依赖性属性值; 功能性的可依赖性需求是指避免、

侦测、容忍系统故障和失效或从中恢复的功能性需求。

dependability case (可依赖性案例)。一种用于支撑系统开发者关于系统可依赖性的声明的结构化文档。安全性案例和信息安全案例是可依赖性案例的两种特殊类型。

design pattern (设计模式)。对于通用问题的一个经过成功验证的解决方案, 以一种可复用的方式描述了其中的经验和好的实践。这是一种可以以多种不同方式进行实例化的抽象表示。

digital learning environment (数字化学习环境)。用于支持学习的一组集成的软件工具、教育应用和内容。

distributed system (分布式系统)。一种软件系统, 其软件子系统或构件在不同的处理器上执行。

domain (领域)。软件系统使用所处的特定问题或业务范围。领域的例子包括实时控制、业务数据处理、电信交换。

domain model (领域模型)。领域抽象 (例如政策、规程、对象、关系、事件) 的一种定义。它可以作为某些问题领域的知识基础。

DSDM (Dynamic System Development Method, 动态系统开发方法)。据称是最早的敏捷开发方法之一。

embedded system (嵌入式系统)。嵌入在硬件设备当中的软件系统, 例如在手机里的软件系统。嵌入式系统通常是实时系统, 因此必须对环境中的事件做出及时的响应。

emergent property (涌现特性)。只有当系统的所有构件都已经集成在一起创建系统时才会表现的一种属性。

Enterprise Java Beans (EJB, 企业 Java Beans)。一种基于 Java 的构件模型。

enterprise resource planning (ERP) system (企业资源规划系统)。这是一种大规模软件系统, 包括支持业务企业运行的一系列能力并提供了跨越这些能力的信息共享手段。例如, 一个 ERP 系统可能包括对供应链管理、制造和分发的支持。ERP 系统需要按照使用该系统的企业的需求进行配置。

ethnography (人种学)。一种可以用于需求抽取

- 盒分析的观察技术。人种学学者自己深入到用户环境中并观察他们的日常工作习惯。对于软件支持的需求可以从这些观察中推导出来。
- event-based systems** (基于事件的系统)。一类系统,这类系统的运行控制由系统环境中生成的事件来决定。绝大多数实时系统都是基于事件的系统。
- extreme programming** (XP, 极限编程)。一种广泛使用的敏捷软件开发方法,所包含的实践有基于情景的需求、测试先行的开发和结对编程。
- fault avoidance** (故障避免)。通过某种方式开发软件以使得故障不会被引入软件中。
- fault detection** (故障检测)。在故障导致系统失效前使用过程和运行时检查来发现并移除程序中的故障。
- fault tolerance** (容错)。系统在故障发生后仍然保持运行的能力。
- fault-tolerant architectures** (容错体系结构)。被设计用于支持从软件故障中恢复的系统结构。这些体系结构都是基于冗余和多样性的软件构件。
- formal methods** (形式化方法)。一种软件开发的方法,其中软件使用形式化的数学符号(例如断言和集合)进行建模。形式化的转换将这些模型转换为代码。主要用于关键性系统的规格说明和开发。
- Gantt chart** (甘特图)。见 bar chart。
- Git**。一种分布式的版本管理和系统构建工具,其中开发者拥有项目存储库的完整拷贝使得他们可以同时工作。
- GitHub**。一个维护了大量 Git 存储库的服务器。这些存储库可以是私有或公共的。有许多开源项目的存储库都在 GitHub 上。
- hazard** (危险)。系统中的某种可能导致事故的情形或状态。
- host-target development** (针对目标主机的开发)。一种软件开发模式,其中软件在一台独立的计算设备上开发,所开发的软件将在这台计算设备上运行。嵌入式和移动系统一般采取这种开发方法。
- iLearn system** (iLearn 系统)。一个支持校内学习的数字学习环境。在本书中被用作一个案例研究。
- incremental development** (增量式开发)。一种软件开发方法,其中软件以增量的方式进行交付和部署。
- information hiding** (信息隐藏)。利用编程语言所提供的机制隐藏数据结构的表示并控制对这些结构的外部访问。
- inspection** (审查)。见 program inspection。
- insulin pump** (胰岛素泵)。一种软件控制的医疗设备,对糖尿病患者注射控制剂量的胰岛素。在本书中被用作一个案例研究。
- integrated application system** (集成的应用系统)。通过集成两个或更多的可配置应用系统或遗留系统创建的应用系统。
- interface** (接口)。与软件构件相关的属性和操作的规格说明。接口是作为访问构件功能的一种手段使用的。
- ISO 9000/9001**。由国际标准化组织 (ISO) 定义的一组质量管理过程标准。ISO 9001 是最适用于软件开发的 ISO 标准。这些标准可以用于认证一个组织中的质量管理过程。
- iterative development** (迭代式开发)。一种软件开发方法,其中规格说明、设计、编码和测试过程是交替进行的。
- J2EE**。是 Java 2 平台企业版 (Java 2 Platform Enterprise Edition) 的缩写。这是一个复杂的中间件系统,支持使用 Java 开发基于构件的 Web 应用。它包括一个面向 Java 构件、API、服务的构件模型。
- Java** (Java 语言)。一种广泛使用的面向对象编程语言,是由 SUN 公司 (现在是 Oracle) 以平台独立性为目的设计的。
- language processing system** (语言处理系统)。一个将一种语言翻译为另一种语言的系统。例如,编译器就是一个将程序源代码翻译成目标代码的语言处理系统。
- legacy system** (遗留系统)。一种社会技术系统,它对于某个组织是有用的或必需的,但却是用过时的技术或方法开发的。由于遗留系统经常执行一些关键的业务功能,因此不得不对它们进行维护。
- Lehman's laws** (Lehman 定律)。针对影响复杂

- 软件系统演化的因素的一组假设。
- maintenance** (维护)。系统投入运行后对其进行变更和修改的过程。
- mean time to failure (MTTF, 平均失效间隔时间)**。观察到的两次系统失效之间的平均时间。在可靠性规格说明中使用。
- Mentcare system (Mentcare 系统)**。心理健康病人护理管理系统。这是一个用于记录针对有心理健康问题的病人的诊断和治疗信息的系统。在本书中被用作一个案例研究。
- middleware (中间件)**。分布式系统中的基础软件。中间件帮助管理系统中的分布式实体以及系统数据库之间的交互。中间件的例子包括对象请求代理以及事务管理系统。
- misuse case (滥用案例)**。针对与一个系统用况相关的可能的系统攻击的描述。
- model-driven architecture (MDA, 模型驱动的体系结构)**。在构建一组系统模型的基础上开发软件的方法, 这些模型可以被自动或半自动化地处理以生成一个可执行的系统。
- model checking (模型检查)**。一种静态验证方法, 其中会对系统的状态模型进行彻底的分析以发现不可达的状态。
- model-driven development (MDD, 模型驱动的开发)**。围绕使用 UML 表达的系统模型而不是使用编程语言表达的代码的一种软件工程方法。它对 MDA 进行了扩展以考虑除了开发之外的其他活动 (例如需求工程和测试)。
- multi-tenant databases (多租户数据库)**。一种数据库, 其中来自多个不同组织的信息被存放在同一个数据库中。用于软件即服务 (SaaS) 的实现中。
- mutual exclusion (互斥)**。一种机制, 保证并发进程保持对存储器的控制直到更新或访问完成。
- .NET**。一种规模很大、用于开发微软 Windows 应用的框架。其中包括定义了 Windows 系统中的构件以及相关的支持构件运行的中间件的标准的构件模型。
- object class (对象类)**。一个对象类定义了对对象的属性和操作。对象在运行时通过实例化类定义来创建。在一些面向对象语言中, 对象类名可以用作类型名。
- object model (对象模型)**。一种软件系统模型, 由一组对象类以及这些类之间的关系组成。针对对象模型存在多种不同的观点, 例如基于状态的观点以及基于序列的观点。
- Object-Oriented (OO) development (面向对象的开发)**。一种软件开发方法, 其中系统中的基本抽象是独立的对象。在规格说明、设计和开发中采用同一种抽象。
- object constraint language (OCL, 对象约束语言)**。作为 UML 的一部分的一种语言, 用于定义适用于 UML 模型中的对象类和交互的断言。使用 OCL 来定义构件是模型驱动开发的一个基本部分。
- Object Management Group (OMG, 对象管理组)**。由许多公司组成的组织, 目的是制定面向对象开发的标准。由 OMG 提出的标准包括 CORBA、UML 和 MDA 等。
- open source (开源)**。一种软件开发方法, 其中系统的源代码是公开的, 鼓励外部用户参与到系统的开发当中来。
- operational profile (运行概况)**。一组人工的系统输入, 反映了一个运行系统中处理的输入的模式。用于可靠性测试。
- pair programming (结对编程)**。一种开发情形, 其中程序员两两配对对工作而不是独立工作来开发代码。它是极限编程的一个基本部分。
- peer to peer system (对等系统)**。一种分布式系统, 其中没有区分客户端和服务端。系统中的计算机可以同时作为客户端和服务端。对等系统的应用包括文件共享、即时通信、合作支持系统。
- People Capability Maturity Model (P-CMM, 人员能力成熟度模型)**。一种过程成熟度模型, 反映了一个机构在管理机构中的人员的技能、培训和经验方面的有效性。
- plan-driven process (计划驱动的过程)**。一种软件过程, 其中所有的过程活动都是在软件开发之前计划好的。
- planning game (计划游戏)**。一种以对于实现用户故事所需的时间的估算为基础的项目计划方法。用于一些敏捷方法中。
- predictor metric (预测性量度)**。一种软件量度, 用作对软件系统的一些特性 (例如可靠性或

可维护性)进行预测的基础。

probability of failure on demand (POFOD, 按需失效概率) 一种基于一个软件系统在请求其服务时失效的可能性的可靠性度量。

process improvement (过程改进) 一种对软件开发过程的改变,目的是使过程更有效率或提高过程输出的质量。例如,如果你的目标是减少所交付的软件中的缺陷数量,那么你可以通过增加新的确认活动来改进过程。

process model (过程模型) 过程的一种抽象表示。过程模型可能从不同的视角来创建,可以显示一个过程中所包含的活动、过程中所使用的制品、适用于该过程的约束,以及参与过程的人所扮演的角色。

process maturity model (过程成熟度模型) 一种针对一个过程包含好的实践以及与过程改进相适应的度量、自我反省与改进能力的程度的模型。

program evolution dynamics(程序演化动力学) 针对演化中的软件系统的变化方式的研究。有人认为 Lehman 定律主宰着程序演化的动力学。

program generator (程序生成器) 从高层的抽象规格说明生成另一个程序的程序。生成器中包含着可以在每次生成活动中复用的知识。

program inspection(程序审查) 一种评审过程,其中一组审查人员以发现程序错误为目的逐行检查一个程序。程序审查经常是由一个常见编程错误的检查表驱动的。

Python (Python 语言) 一种具有动态类型编程语言,特别适合于开发基于 Web 的系统。

quality management (QM, 质量管理) 一组过程,关注定义如何实现高质量的软件以及开发软件的组织如何知道软件满足所需的质量要求。

quality plan (质量规划) 定义所需要使用的质量过程和规程的计划。其中包括为产品和过程选择并实例化标准,定义最为重要的系统质量属性。

rapid application development (RAD, 快速应用开发) 一种关注快速交付软件的软件开发方法。该方法经常使用数据库编程以及开

发支持工具(例如界面和报表生成器)。

rate of occurrence of failure (ROCOF, 失效发生率) 基于在一个给定时间段内观察到的系统失效次数的可靠性度量。

Rational Unified Process (RUP, Rational 统一过程) 一种通用的软件过程模型,其中将软件开发表示为一个四阶段的迭代活动,即初始、细化、构造、转换。初始阶段为系统建立一个业务案例,细化阶段定义体系结构,构造阶段实现系统,而转换阶段在客户环境中部署系统。

real time system (实时系统) 一个必须实时识别并处理外部事件的系统。这类系统的正确性不仅取决于它做什么,而且还取决于做这些事情的速度。实时系统通常由一组并发的进程组成。

reductionism (还原论) 一种工程方法,其基本思想是将问题分解为子问题,独立解决这些子问题,然后再通过集成这些解决方案来创建针对更大问题的解决方案。

reengineering (再工程) 修改软件系统以使其更容易理解和变更。再工程经常包括软件和数据的重组和重构、程序简化和再文档化。

reengineering, business process (再工程, 业务过程) 改变业务过程以使之满足新的组织目标,例如降低成本以及加快执行速度。

refactoring (重构) 修改一个程序以改进它的结构和可读性,同时不改变其功能。

reference architecture (参考体系结构) 一种通用、理想的体系结构,其中包含系统可能包括的所有特征。通过这种方式可以告诉设计者某类系统的通用结构是什么,而不仅仅是作为创建特定系统体系结构的基础。

release (发布版本) 提供给系统客户的一个软件系统的版本。

reliability (可靠性) 系统提供所指定的服务的能力。可靠性可以被定量地刻画为接受请求时的失效概率或者失效发生率。

reliability growth modeling (可靠性增长建模) 针对一个系统如何随着测试以及程序缺陷的移除而发生可靠性的变化(改进)进行模型的开发。

requirement, functional (需求, 功能性的) 对

一个系统中应当实现的一些功能或特征的一种陈述。

requirement, non functional (需求, 非功能性的)。适用于一个系统的一种约束或期望行为的陈述。约束可以针对正在开发的软件的涌现性特性或者针对开发过程。

requirement management (需求管理)。管理对需求的变更以保证所做的变更是经过适当的分析并在系统中保持追踪的过程。

resilience (韧性)。对于系统能够在多大程度上在出现破坏性事件(例如设备失效和网络攻击)的情况下保持关键性服务的持续性的一种判断。

REST (Representational State Transfer)。是一种基于简单的使用 HTTP 协议通信的客户端/服务器交互的开发风格。REST 基于可识别的资源(具有 URI)的理念。与资源的所有交互都是基于 HTTP POST、GET、PUT 和 DELETE 进行的。目前被广泛用于实现低开销的 Web 服务 (RESTful 服务)。

revision control systems (修订控制系统)。见 version control systems。

risk (风险)。一种不希望看到的、会对某些目标的实现构成威胁的结果。一个过程风险威胁着过程的进度或成本; 一个产品风险则意味着系统的某些需求可能无法满足。一个安全性风险是对一个危害可能导致一个事故的可能性的度量。

risk management (风险管理)。识别风险、评估风险的严重性、规划风险应对措施、监控软件和软件过程中的风险的过程。

Ruby (Ruby 语言)。一种具有动态类型的编程语言, 特别适合于 Web 应用编程。

SaaS (软件即服务)。见 software as a service。

safety (安全性)。一个系统在运行过程中不出现可能导致人身伤害、人员死亡或系统环境破坏的行为的能力。

safety case (安全案例)。一种证明一个系统是安全的且可以保障信息安全的一套证据和结构化的论据。很多关键性系统必须具备相关的安全案例, 并由外部监管者在系统获得使用许可之前评估和审批。

SAP。一家德国公司, 开发了一个著名的且被广

泛使用的 ERP 系统。也可以指对该 ERP 系统自身的命名。

scenario (场景)。对于一个系统的典型使用方式或者一个用户执行某些活动的典型方式的一种描述。

scenario testing (场景测试)。一种软件测试方法, 其中测试用例是从系统使用场景中派生出来的。

Scrum (Scrum 方法)。一种敏捷开发方法, 基于一种被称为冲刺 (sprint) 的短开发迭代周期。Scrum 可以与其他敏捷方法(例如 XP)一起用作敏捷项目管理的基础。

security (信息安全性)。系统保护自身免于偶然的和恶意的入侵的能力。信息安全性包括机密性、完整性和可用性。

SEI。是 Software Engineering Institute 的缩写。是一个软件工程研究和技术转化中心, 其创建的目的是改进美国企业中的软件工程标准。

sequence diagram (顺序图)。描述为完成某些操作所需要的交互序列的图。在 UML 中, 顺序图可以与用况相关联。

server (服务器)。向其他(客户端)程序提供服务的程序。

service (服务)。见 Web service。

socio-technical system (社会技术系统)。是一种包含软硬件构件、定义了操作人员需要遵循的运行过程并在一个组织内部运行的系统。因此, 这类系统受组织的政策、规程及结构的影响。

software analytics (软件解析学)。对于与软件系统相关的动态数据和静态数据进行自动分析以发现这些数据之间的关系。这些关系可以提供关于改进软件质量的可能途径的深入洞悉。

software architecture (软件体系结构)。描述软件系统基本结构和组织的模型。

software as a service (SaaS, 软件即服务)。通过 Web 浏览器远程访问而不是在本地计算机上安装的软件应用。越来越多地被用于向最终用户提供应用服务。

software life cycle (软件生命周期)。经常作为软件过程的另一个名称。最初被用于指软件

- 过程中的瀑布模型。
- software metric (软件量度)**。软件系统或过程的属性,可以用数字化的方式表达并且可以被测量。过程量度是过程的属性,例如完成任务的时间;产品量度是软件自身的属性,例如规模或复杂度。
- software process (软件过程)**。在软件系统的开发和演化中所涉及的活动及过程。
- software product line(软件产品线)**。见 application family。
- spiral model (螺旋模型)**。一种开发过程模型,其中过程被表示为一个螺旋,螺旋的每一圈包含过程的不同阶段。当从螺旋的一个圈移动到另一个圈时,就重复了该过程的所有阶段。
- state diagram (状态图)**。一种 UML 图,其中描述了系统的状态以及触发系统从一个状态转换到另一个状态的事件。
- static analysis (静态分析)**。基于工具对程序源代码进行分析以发现错误和异常。异常(例如对一个中间没有使用过的变量连续进行赋值)可能是编程错误的指示器。
- structured method (结构化方法)**。一种软件设计方法,定义应当要开发的系统模型、适用于这些模型的规则和指南,以及在进行设计过程中应遵循的过程。
- Structured Query Language (SQL, 结构化查询语言)**。用于关系数据库编程的一种标准语言。
- Subversion**。一种广泛使用的开源版本控制和系统构建工具,可在很多平台上使用。
- Swiss cheese model (瑞士奶酪模型)**。一种针对操作人员错误或网络攻击的系统防护模型,考虑了不同防护层上的漏洞。
- system (系统)**。一个系统是一组相互关联、不同种类的构件的一种有目的的组合,这些构件一起工作以向系统所有者和用户提供一组服务。
- system building (系统构建)**。编译组成一个系统的构件或单元并将这些与其他构件相链接以创建一个可执行程序的过程。系统构建通常是自动化的,这样需要重新编译的部分可以最小化。这种自动化构建可以作为语言处理系统的内部机制(如 Java 中所做的),或者使用软件工具来支持系统构建。
- systems engineering (系统工程)**。关于系统规格说明、系统构件集成、确保系统满足其需求的系统测试的过程。系统工程关注整个社会技术系统——软件、硬件及运行过程,而不仅是系统软件。
- system of systems (系统之系统)**。通过集成两个或更多的现有系统创建的系统。
- system testing (系统测试)**。在交付给客户之前对一个完整系统的测试。
- test coverage (测试覆盖)**。测试整个系统代码的系统测试的有效性。一些公司有相应的测试覆盖标准,例如系统测试应当保证所有程序语句至少被执行一次。
- test-driven development (测试驱动的开发)**。一种软件开发方法,其中可执行的测试是在程序代码之前编写的。每次对程序进行修改后都会自动执行这一组测试。
- TOGAF**。一种由对象管理组(OMG)支持的体系结构框架,其目的是支持面向系统之系统的企业体系结构的开发。
- transaction (事务)**。一个与计算机系统的交互单元。事务是独立的和原子的(它们不能被分割成更小的单元),并且是恢复、一致性及并发性的基本单元。
- transaction processing system(事务处理系统)**。一种确保不同事务的处理过程不会相互干扰并且单个事务的失效不会影响到其他事务及系统数据的系统。
- Unified Modeling Language (UML, 统一建模语言)**。一种用于面向对象开发的图形化语言,其中包含描述一个系统的不同视图的多种类型的系统模型。UML 已经成为面向对象建模的事实标准。
- unit testing (单元测试)**。由软件开发者或开发团队所做的对单个程序单元的测试。
- use case (用例)**。与系统的一种交互类型的规格说明。
- use-case diagram (用例图)**。一种用于识别用例并图形化地描述所涉及的用户 UML 图。用例图必须通过补充附加的信息来完整地描述用例。
- user interface design (用户界面设计)**。设计系

统用户访问系统功能及显示系统所产生的信息的方式的过程。

user story (用户故事)。解释一个软件或系统如何被使用的情形以及关于与系统之间可能发生的交互的一种自然语言描述。

validation (确认)。检查系统是否满足了客户的需要和期望的过程。

verification (验证)。检查一个系统是否符合其规格说明的过程。

version control (版本控制)。管理对于一个软件系统及其构件的变更的过程,其目的是了解构件/系统的每一个版本中进行了哪些修改以及恢复/重建该构件/系统此前的版本。

version control (VC) systems (版本控制系统)。用于支持版本控制过程的软件工具。可以基于集中式的或者分布式的存储库。

waterfall model(瀑布模型)。一种软件过程模型,包含规格说明、设计、实现、测试及维护这几个离散的开发阶段。原则上,一个阶段必须在进入下一个阶段之前完成。在实践中,阶段之间存在显著的迭代。

Web service (Web 服务)。可以使用标准协议通过互联网访问的独立的软件构件。完全自包含,没有任何外部依赖。针对 Web 服务开发了基于 XML 的标准,例如用于 Web 服务信息交换的 SOAP (标准对象访问协议)、用于 Web 服务接口定义的 WSDL (Web 服务

定义语言)。然而,REST 方法也可以用于 Web 服务实现。

white-box testing (白盒测试)。一种程序测试方法,其中测试是基于程序及其构件的结构信息的。能够访问到源代码是白盒测试的基本要求。

wicked problem (非常规问题)。由于问题的相关元素之间交互的复杂性而导致无法完整刻画或理解的问题。

wilderness weather system (野外气象系统)。一个收集偏远地区气象状况数据的系统。在本书中被用作案例研究。

workflow (工作流)。为了完成某个任务而对一个业务过程的详细的定义。工作流通常用图形化的方式表达,显示各个过程活动以及由每个活动产生和消费的信息。

WSDL。一种基于 XML 的符号系统,用于定义 Web 服务的接口。

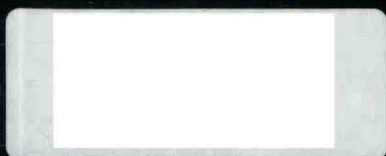
XML (Extended Markup Language, 可扩展标记语言)。可扩展标记语言。XML 是一种支持结构化数据交换的文本标记语言。每个数据域用标记符界定,标记符给出该数据域的信息。如今 XML 已经得到了广泛应用并且成为 Web 服务协议的基础。

XP。见 Extreme Programming。

Z (Z 方法)。一种基于模型的形式化规格说明语言,是由英国牛津大学开发的。

软件工程 原书第10版

Software Engineering Tenth Edition



本书是软件工程领域的经典教材，自1982年第1版出版至今，伴随着软件工程学科的发展不断更新，影响了一代又一代的软件工程人才，对学科建设也产生了积极影响。全书共四个部分，完整讨论了软件工程各个阶段的内容，适合软件工程相关专业本科生和研究生学习，也适合软件工程师参考。

新版重要更新

- 全面更新了关于敏捷软件工程的章节，增加了关于Scrum的新内容。此外还根据需要对其他章节进行了更新，以反映敏捷方法在软件工程中日益增长的应用。
- 增加了关于韧性工程、系统工程、系统之系统的新章节。
- 对于涉及可靠性、安全、信息安全的三章进行了重新组织。
- 在第18章“面向服务的软件工程”中增加了关于RESTful服务的新内容。
- 更新和修改了关于配置管理的章节，增加了关于分布式版本控制系统的新内容。
- 将关于面向方面的软件工程以及过程改进的章节移到了本书的配套网站（software-engineering-book.com）上。
- 在网站上新增了补充材料，包括一系列教学视频。

作者简介

伊恩·萨默维尔（Ian Sommerville）英国著名软件工程专家，曾任圣安德鲁斯大学软件工程系教授，2014年退休。他在软件工程的教学和科研方面有40多年的经验，研究领域包括需求工程、大规模复杂系统和系统可靠性等。他撰写的软件工程教材畅销全球，销量超过75万册。



华章教育服务微信号



www.pearson.com



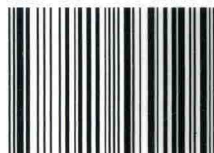
华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

010-88379604

上架指导: 计算机\软件工程

ISBN 978-7-111-58910-5



9 787111 589105 >

定价: 89.00元